Document Number:
 N3337

 Date:
 2012-01-16

 Revises:
 N3291

Reply to: Stefanus Du Toit

Intel Corporation cxxeditor@gmail.com

# Working Draft, Standard for Programming Language C ++

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

## Contents

Contents

C	Contents			
Li	st of T	Cables	x	
Li	st of F	ligures	xiv	
1	Gene 1.1 1.2 1.3 1.4 1.5 1.6 1.7	Scope Normative references Terms and definitions Implementation compliance Structure of this International Standard Syntax notation The C++ memory model The C++ object model	1 1 2 5 5 6 6 7	
	1.9 1.10 1.11	Program execution	8 11 14	
2	Lexic 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 2.10 2.11 2.12 2.13 2.14	al conventions Separate translation Phases of translation Character sets Trigraph sequences Preprocessing tokens Alternative tokens Tokens Comments Header names Preprocessing numbers Identifiers Keywords Operators and punctuators Literals	16 16 17 18 19 20 20 20 21 21 22 22 23	
3	Basic 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 3.10	concepts Declarations and definitions One definition rule Scope Name lookup Program and linkage Start and termination Storage duration Object lifetime Types Lvalues and rvalues	32 34 36 42 55 58 62 65 69 74	

ii

	3.11	Alignment
4	Stand	lard conversions 77
	4.1	Lvalue-to-rvalue conversion
	4.2	Array-to-pointer conversion
	4.3	Function-to-pointer conversion
	4.4	Qualification conversions
	4.5	Integral promotions
	4.6	Floating point promotion
	4.7	Integral conversions
	4.8	Floating point conversions
	4.9	Floating-integral conversions
	4.10	Pointer conversions
	4.11	Pointer to member conversions
	4.12	Boolean conversions
	4.13	Integer conversion rank
5	-	essions 83
	5.1	Primary expressions
	5.2	Postfix expressions
	5.3	Unary expressions
	5.4	Explicit type conversion (cast notation)
	5.5	Pointer-to-member operators
	5.6	Multiplicative operators
	5.7	Additive operators
	5.8	Shift operators
	5.9	Relational operators
	5.10	Equality operators
	5.11	Bitwise AND operator
	5.12	Bitwise exclusive OR operator
	5.13	Bitwise inclusive OR operator
	5.14	Logical AND operator
	5.15	Logical OR operator
	5.16	Conditional operator
	5.17	Assignment and compound assignment operators
	5.18	Comma operator
	5.19	Constant expressions
6		ments 124
	6.1	Labeled statement
	6.2	Expression statement
	6.3	Compound statement or block
	6.4	Selection statements
	6.5	Iteration statements
	6.6	Jump statements
	6.7	Declaration statement
	6.8	Ambiguity resolution
-	י ח	
7		rations 133
	7.1	Specifiers
	7.2	Enumeration declarations

© ISO	/IEC	N3337

	7.3	Namespaces	2
	7.4	The asm declaration	4
	7.5	Linkage specifications	4
	7.6	Attributes	7
8	Decla		
	8.1	Гуре names	
	8.2	Ambiguity resolution	4
	8.3	Meaning of declarators	5
	8.4	Function definitions	7
	8.5	Initializers	0
9	Class		
	9.1	Class names	6
	9.2	Class members	7
	9.3	Member functions	0
	9.4	Static members	3
	9.5	Unions	4
	9.6	Bit-fields	
	9.7	Nested class declarations	
	9.8	Local class declarations	
	9.9	Nested type names	
	0.0	tropod type names	
10	Deriv	ed classes 21	9
	10.1	Multiple base classes	0
	10.2	Member name lookup	
	10.3	Virtual functions	
	10.4	Abstract classes	
	10.1	22	Ĭ
11	Meml	er access control 23	1
	11.1	Access specifiers	2
	11.2	Accessibility of base classes and base class members	
	11.3	Friends	
	11.4	Protected member access	
	11.5	Access to virtual functions	
	11.6	Multiple access	
	11.7	Nested classes	
	11.1	Nested Classes	U
12	Speci	l member functions 24	2
	12.1	Constructors	
	12.2	$\begin{array}{llllllllllllllllllllllllllllllllllll$	
	12.3	Conversions	
		Destructors	
	12.4		
	12.5	Free store	
	12.6	Initialization	
	12.7	Construction and destruction	
	12.8	Copying and moving class objects	
	12.9	Inheriting constructors	9
10		11	•
13	Overl	8	
	13.1	Overloadable declarations	3

© ISO	/IEC	N3337

13.2 13.3 13.4 13.5 13.6	Declaration matching	76 94 96
14 Tem		03
14.1	Template parameters	
14.2	Names of template specializations	07
14.3	Template arguments	08
14.4	Type equivalence	14
14.5	Template declarations	15
14.6	Name resolution	31
14.7	Template instantiation and specialization	45
14.8	Function template specializations	56
		<b>77</b>
15.1	Throwing an exception	
15.2	Constructors and destructors	
15.3	Handling an exception	
15.4	Exception specifications	
15.5	Special functions	85
16 Pren	rocessing directives 3	87
16.1	Conditional inclusion	
16.2	Source file inclusion	
16.3	Macro replacement	
16.4	Line control	
16.5	Error directive	
16.6	Pragma directive	
16.7	Null directive	
16.8	Predefined macro names	
16.9	Pragma operator	
	0	•
17 Libra	J	98
17.1	General	
17.2	The C standard library	99
17.3	Definitions	99
17.4	Additional definitions	-
17.5	Method of description (Informative)	
17.6	Library-wide requirements	07
10 T	4 191	00
_	6	<b>26</b>
18.1	General	
18.2	Types	
18.3	Implementation properties	
18.4	Integer types	
18.5	Start and termination	
18.6	Dynamic memory management	
18.7	Type identification	
18.8	Exception handling	41

$\odot$ ISO	O/IEC	N3337
$(\circ)$ $\mathbf{I}$ $\mathbf{D}$ $\mathbf{V}$	O/IEC	110001

	18.9 18.10	Initializer lists				
10	19 Diagnostics library 456					
10	19.1	General				
	19.2	Exception classes				
	19.3	Assertions				
	19.4	Error numbers				
	19.5	System error support				
	10.0	System of or support	100			
<b>20</b>		ral utilities library	<b>472</b>			
	20.1	General				
	20.2	Utility components				
	20.3	Pairs				
	20.4	Tuples				
	20.5	Class template bitset				
	20.6	Memory				
	20.7	Smart pointers				
	20.8	Function objects				
	20.9	Metaprogramming and type traits				
	20.10	Compile-time rational arithmetic				
		Time utilities				
		Class template scoped_allocator_adaptor				
	20.13	Class type_index	599			
21	String	gs library	601			
	21.1	General	601			
	21.2	Character traits				
	21.3	String classes				
	21.4	Class template basic_string				
	21.5	Numeric conversions				
	21.6	Hash support				
	21.7	Null-terminated sequence utilities				
22		ization library	645			
	22.1	General				
	22.2	Header <locale> synopsis</locale>				
	22.3	Locales				
	22.4	Standard locale categories				
	22.5	Standard code conversion facets				
	22.6	C library locales	700			
23	Conta	ainers library	<b>702</b>			
	23.1	General	702			
	23.2	Container requirements				
	23.3	Sequence containers				
	23.4	Associative containers				
	23.5	Unordered associative containers				
	23.6	Container adaptors				
24	Iterat	tors library	803			

Contents vi

	24.1	General	803
	24.2	Iterator requirements	803
	24.3	Header <iterator> synopsis</iterator>	808
	24.4	Iterator primitives	811
	24.5	Iterator adaptors	815
	24.6	Stream iterators	
25	Algor	rithms library	836
	25.1	General	
	25.2	Non-modifying sequence operations	
	25.3	Mutating sequence operations	
	25.4	Sorting and related operations	
	25.5	C library algorithms	
		erics library	874
	26.1	General	
	26.2	Numeric type requirements	
	26.3	The floating-point environment	
	26.4	Complex numbers	
	26.5	Random number generation	
	26.6	Numeric arrays	
	26.7	Generalized numeric operations	
	26.8	C library	955
<b>27</b>	Input	output library	960
	27.1	General	960
	27.2	Iostreams requirements	
	27.3	Forward declarations	961
	27.4	Standard iostream objects	963
	27.5	Iostreams base classes	965
	27.6	Stream buffers	984
	27.7	Formatting and manipulators	994
	27.8	String-based streams	1021
	27.9	File-based streams	1032
26	Pogu	lar expressions library	1048
	28.1	General	
	28.2	Definitions	
	28.3	Requirements	
	28.4	Header <regex> synopsis</regex>	
	28.5	Namespace std::regex_constants	
	28.6	. 9 -	
		Class regex_error	
	28.7	Class template regex_traits	
	28.8	Class template basic_regex	
	28.9	Class template sub_match	
	28.10	Class template match_results	
	28.11	Regular expression algorithms	
	28.12	Regular expression iterators	
	28.13	Modified ECMAScript regular expression grammar	1093
<b>29</b>	Atom	ic operations library	1096

	29.1 29.2	General	
	29.3	Order and consistency	
	29.4	Lock-free property	
	29.4 $29.5$	Atomic types	
	29.6	Operations on atomic types	
	29.0 $29.7$	Flag type and operations	
	29.7	Fences	
	29.0	rences	1112
30		ad support library	1114
	30.1	General	
	30.2	Requirements	
	30.3	Threads	
	30.4	Mutual exclusion	
	30.5	Condition variables	1135
	30.6	Futures	1144
Α	Gran	nmar summary	1161
	A.1	Keywords	
	A.2	Lexical conventions	
	A.3	Basic concepts	
	A.4	Expressions	
	A.5	Statements	
	A.6	Declarations	
	A.7	Declarators	
	A.8	Classes	
	A.9	Derived classes	
	A.10	Special member functions	
	A.11	Overloading	
	A.12	Templates	
	A.13	Exception handling	
	A.14	Preprocessing directives	
		•	
В	Imple	ementation quantities	1181
$\mathbf{C}$	Comp	patibility	1183
	C.1	C++ and ISO C	
	C.2	C++ and ISO C++ 2003	
	C.3	C standard library	1198
D	Com	patibility features	1202
	D.1	Increment operator with bool operand	1202
	D.2	register keyword	
	D.3	Implicit declaration of copy functions	
	D.4	Dynamic exception specifications	
	D.5	C standard library headers	
	D.6	Old iostreams members	
	D.7	char* streams	
	D.8	Function objects	
	D.9	Binders	
	D.10	auto_ptr	

©	ISO/I	EC N	3337
	D.11	Violating exception-specifications	. 1221
$\mathbf{E}$	Unive E.1 E.2	Ranges of characters disallowed initially	
$\mathbf{F}$	Cross	s references	1223

$\mathbf{E}$	Unive	ersal character names for identifier characters	1222
	E.1	Ranges of characters allowed	1222
	E.2	Ranges of characters disallowed initially $\hdots$	1222
F	Cross	references	1223
Ind	ndex		
Inc	ndex of grammar productions		
In	dex of	library names	1272
In	dex of	implementation-defined behavior	1308

Contentsix

## List of Tables

1	Trigraph sequences	18
2	Alternative tokens	20
3	Identifiers with special meaning	22
4	Keywords	22
5	Alternative representations	22
6	Types of integer constants	24
7	Escape sequences	
8	String literal concatenations	29
9	Relations on const and volatile	74
10	simple-type-specifiers and the types they specify	146
11	Relationship between operator and function call notation	
12	Conversions	288
13	Library categories	
14	C++ library headers	
15	C++ headers for C library facilities	
16	C++ headers for freestanding implementations	
17	EqualityComparable requirements	
18	LessThanComparable requirements	
19	DefaultConstructible requirements	410
20	MoveConstructible requirements	410
21	CopyConstructible requirements (in addition to MoveConstructible)	411
22	MoveAssignable requirements	411
23	CopyAssignable requirements (in addition to MoveAssignable)	411
24	Destructible requirements	
25	NullablePointer requirements	
26	Hash requirements	
27	Descriptive variable definitions	
28	Allocator requirements	
29	Language support library summary	426
30	Header <cstddef> synopsis</cstddef>	
31	Header <climits> synopsis</climits>	
32	Header <cfloat> synopsis</cfloat>	
33	Header <cstdlib> synopsis</cstdlib>	
34	v ·	
	Header <csetjmp> synopsis</csetjmp>	
$\frac{35}{26}$	Header <csignal> synopsis</csignal>	
36	Header <cstdalign> synopsis</cstdalign>	
37	Header <cstdarg> synopsis</cstdarg>	
38	Header <cstdbool> synopsis</cstdbool>	
39	Header <cstdlib> synopsis</cstdlib>	
40	Header <ctime> synopsis</ctime>	455

List of Tables x

42	Diagnostics library summary	
	Header <cassert> synopsis</cassert>	
43	Header <cerrno> synopsis</cerrno>	61
44	General utilities library summary	72
45	Header <cstdlib> synopsis</cstdlib>	
46	Header <cstring> synopsis</cstring>	12
47	Primary type category predicates	62
48	Composite type category predicates	63
49	Type property predicates	63
50	Type property queries	668
51	Type relationship predicates	
52	Const-volatile modifications	
53	Reference modifications	
54	Sign modifications	
55	Array modifications	
56	Pointer modifications	
57	Other transformations	
58	Expressions used to perform ratio arithmetic	
59	Clock requirements	
60	Header <ctime> synopsis</ctime>	
00	Theader Ctimes synopsis	192
61	Strings library summary	01
62	Character traits requirements	
63	basic_string(const Allocator&) effects	
64	basic_string(const basic_string&) effects	
65	basic_string(const basic_string&, size_type, size_type, const Allocator&) effects . (	
66	basic_string(const basic_string&, size_type, size_type, const Allocator&) effects	
67	basic_string(const charT*, size_type, const Allocator&) effects	
60	hagis string(size t short sonst Allegatory) offests	
68 60	basic_string(size_t, charT, const Allocator&) effects	17
68 69	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&,	317
69	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518
69 70	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518 518
69 70 71	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518 518
69 70 71 72	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518 518 518
69 70 71 72 73	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518 518 518 533
69 70 71 72 73 74	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518 518 518 533 542
69 70 71 72 73 74 75	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518 518 518 542 542 543
69 70 71 72 73 74 75 76	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518 518 518 518 542 542 543
69 70 71 72 73 74 75 76 77	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518 518 518 518 542 542 543 543
70 71 72 73 74 75 76 77	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects  operator=(const basic_string <chart, allocator="" traits,="">&amp;) effects  operator=(const basic_string<chart, allocator="" traits,="">&amp;) effects  compare() results  Potential mbstate_t data races  Header <cctype> synopsis  Header <cstring> synopsis  Header <cstring> synopsis  Header <cstdlib> synopsis  Header <cstdlib> synopsis</cstdlib></cstdlib></cstring></cstring></cctype></chart,></chart,>	517 518 518 518 542 542 543 543
69 70 71 72 73 74 75 76 77	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518 518 518 542 542 543 543
70 71 72 73 74 75 76 77 78	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects  operator=(const basic_string <chart, allocator="" traits,="">&amp;) effects  operator=(const basic_string<chart, allocator="" traits,="">&amp;) effects  compare() results  Potential mbstate_t data races  Header <cctype> synopsis  Header <cwctype> synopsis  Header <cwctype> synopsis  Header <cwchar> synopsis  Header <cstdlib> synopsis  Header <cuchar> synopsis  Header <cuchar> synopsis  Header <cuchar> synopsis</cuchar></cuchar></cuchar></cstdlib></cwchar></cwctype></cwctype></cctype></chart,></chart,>	517 518 518 518 533 542 543 543 544
69 70 71 72 73 74 75 76 77 78 79	basic_string(const basic_string&, const Allocator&) and basic_string(&sic_string&&, const Allocator&) effects operator=(const basic_string <chart, allocator="" traits,="">&amp;) effects operator=(const basic_string<chart, allocator="" traits,="">&amp;) effects compare() results Potential mbstate_t data races Header <cctype> synopsis Header <cwctype> synopsis Header <cwctype> synopsis Header <cwchar> synopsis Header <cstdlib> synopsis Header <cuchar> synopsis</cuchar></cuchar></cuchar></cuchar></cuchar></cuchar></cstdlib></cwchar></cwctype></cwctype></cctype></chart,></chart,>	517 518 518 518 518 518 518 518 518 518 518
69 70 71 72 73 74 75 76 77 78 79	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects operator=(const basic_string <chart, allocator="" traits,="">&amp;) effects operator=(const basic_string<chart, allocator="" traits,="">&amp;) effects compare() results Potential mbstate_t data races Header <cctype> synopsis Header <cwctype> synopsis Header <cstring> synopsis Header <cstdlib> synopsis Header <cuchar> synopsis</cuchar></cuchar></cuchar></cuchar></cuchar></cuchar></cuchar></cuchar></cuchar></cuchar></cuchar></cstdlib></cstring></cwctype></cctype></chart,></chart,>	517 518 518 518 518 518 518 518 518 518 518
69 70 71 72 73 74 75 76 77 78 79 80 81 82	basic_string(const basic_string&, const Allocator&) and basic_string&&, const Allocator&) effects operator=(const basic_string <chart, allocator="" traits,="">&amp;) effects operator=(const basic_string<chart, allocator="" traits,="">&amp;) effects compare() results Potential mbstate_t data races Header <cctype> synopsis Header <cwctype> synopsis Header <cstring> synopsis Header <cctdlib> synopsis Header <cctdlib> synopsis Header <cuchar> synopsis Header <cuchar> synopsis Header <cctdlib> synopsis</cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cctdlib></cuchar></cuchar></cctdlib></cctdlib></cstring></cwctype></cctype></chart,></chart,>	517 518 518 518 518 518 518 518 518 518 518
70 71 72 73 74 75 76 77 78 79 80 81 82 83	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	518 518 518 518 518 518 518 518 518 518
70 71 72 73 74 75 76 77 78 79 80 81 82 83 84	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	518 518 518 518 518 518 518 518 518 518
70 71 72 73 74 75 76 77 78 79 80 81 82 83	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	517 518 518 518 518 518 518 518 518 518 518
70 71 72 73 74 75 76 77 78 79 80 81 82 83 84	basic_string(const basic_string&, const Allocator&) and basic_string&&, const Allocator&) effects	518 518 518 518 518 518 518 518 518 518
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85	basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects	518 518 518 518 518 518 518 518 518 518

List of Tables xi

88	Floating-point conversions
89	Length modifier
90	Numeric conversions
91	Fill padding
92	do_get_date effects
93	Header <clocale> synopsis</clocale>
94	Potential setlocale data races
95	Containers library summary
96	Container requirements
97	Reversible container requirements
98	Optional container operations
99	Allocator-aware container requirements
100	Sequence container requirements (in addition to container)
101	Optional sequence container operations
102	Associative container requirements (in addition to container)
103	Unordered associative container requirements (in addition to container)
100	Chordered associative container requirements (in addition to container)
104	Iterators library summary
$104 \\ 105$	Relations among iterator categories
106	Iterator requirements
$100 \\ 107$	Input iterator requirements (in addition to Iterator)
107	Output iterator requirements (in addition to Iterator)
109	Forward iterator requirements (in addition to input iterator)
110	Bidirectional iterator requirements (in addition to forward iterator)
111	Random access iterator requirements (in addition to bidirectional iterator) 808
112	Algorithms library summary
	Header <cstdlib> synopsis</cstdlib>
119	Treader Costation symposis
11/	Numerics library summary
115	Seed sequence requirements
116	Uniform random number generator requirements
117	Random number engine requirements
118	
	Random number distribution requirements
119	Header <cmath> synopsis</cmath>
120	Header <cstdlib> synopsis</cstdlib>
191	Input/output library summary
122	fmtflags effects
123	fmtflags constants
124	iostate effects
125	1
126	seekdir effects
127	Position type requirements
128	basic_ios::init() effects
129	basic_ios::copyfmt() effects
130	seekoff positioning
131	newoff values
132	File open modes
	The open modes
133	seekoff effects

List of Tables xii

	Header <cstdio> synopsis   <td< th=""></td<></cstdio>
136	Regular expressions library summary
137	Regular expression traits class requirements
138	syntax_option_type effects
139	regex_constants::match_flag_type effects when obtaining a match against a character con-
	tainer sequence [first,last)
140	error_type values in the C locale
141	match_results assignment operator effects
142	Effects of regex_match algorithm
143	Effects of regex_search algorithm
	Atomics library summary
	atomic integral typedefs
	atomic <inttypes.h> typedefs</inttypes.h>
147	Atomic arithmetic computations
148	Thread support library summary
149	Standard macros
150	Standard values
151	Standard types
152	* <del>-</del>
153	Standard functions
	C headers
155	strstreambuf(streamsize) effects
156	strstreambuf(void* (*)(size_t), void (*)(void*)) effects
157	
158	seekoff positioning
159	newoff values

List of Tables xiii

## List of Figures

1	Expression category taxonomy
2	Directed acyclic graph
3	Non-virtual base
4	Virtual base
5	Virtual and non-virtual base
6	Name lookup
7	Stream position, offset, and size types [non-normative]

List of Figures xiv

1 General [intro]

1.1 Scope [intro.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.

2 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, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

#### 1.2 Normative references

[intro.refs]

- The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.
  - Ecma International, ECMAScript Language Specification, Standard Ecma-262, third edition, 1999.
  - 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 9899:1999/Cor.2:2004(E), Programming languages C, Technical Corrigendum 2
  - ISO/IEC 9899:1999/Cor.3:2007(E), Programming languages C, Technical Corrigendum 3
  - ISO/IEC 9945:2003, Information Technology Portable Operating System Interface (POSIX)
  - ISO/IEC 10646-1:1993, Information technology Universal Multiple-Octet Coded Character Set (UCS)
     Part 1: Architecture and Basic Multilingual Plane
  - 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
- <sup>2</sup> The library described in Clause 7 of ISO/IEC 9899:1999 and Clause 7 of ISO/IEC 9899:1999/Cor.1:2001 and Clause 7 of ISO/IEC 9899:1999/Cor.2:2003 is hereinafter called the *C standard library*.<sup>1</sup>
- <sup>3</sup> The library described in ISO/IEC TR 19769:2004 is hereinafter called the C Unicode TR.
- <sup>4</sup> The operating system interface described in ISO/IEC 9945:2003 is hereinafter called *POSIX*.
- <sup>5</sup> The ECMAScript Language Specification described in Standard Ecma-262 is hereinafter called ECMA-262.

<sup>1)</sup> With the qualifications noted in Clauses 18 through 30 and in C.3, the C standard library is a subset of the C++ standard library.

#### 1.3 Terms and definitions

[intro.defs]

- For the purposes of this document, the following definitions apply.
- <sup>2</sup> 17.3 defines additional terms that are used only in Clauses 17 through 30 and Annex D.
- <sup>3</sup> 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 [defns.argument]

#### argument

actual argument

actual parameter

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

1.3.2 [defns.argument.macro]

#### argument

actual argument

actual parameter

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

1.3.3 [defns.argument.throw]

#### argument

actual argument

actual parameter

<throw expression> the operand of throw

1.3.4 [defns.argument.templ]

#### argument

actual argument

actual parameter

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

1.3.5 [defns.cond.supp]

#### conditionally-supported

program construct that an implementation is not required to support

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

1.3.6 [defns.diagnostic]

#### diagnostic message

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

1.3.7 [defns.dynamic.type]

#### dynamic type

<gl>
 type of the most derived object (1.8) to which the glvalue denoted by a glvalue expression refers [Example: if a pointer (8.3.1) p whose static type is "pointer to class B" is pointing to an object of class D, derived from B (Clause 10), the dynamic type of the expression \*p is "D." References (8.3.2) are treated similarly. — end example]

1.3.8 [defns.dynamic.type.prvalue]

§ 1.3 2

#### dynamic type

cprvalue> static type of the prvalue expression

1.3.9 [defns.ill.formed]

#### ill-formed program

program that is not well formed

1.3.10 [defns.impl.defined]

#### implementation-defined behavior

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

[defns.impl.limits]

#### implementation limits

restrictions imposed upon programs by the implementation

1.3.12 [defns.locale.specific]

#### locale-specific behavior

behavior that depends on local conventions of nationality, culture, and language that each implementation documents

1.3.13 [defns.multibyte]

#### multibyte character

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

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

1.3.14 [defns.parameter]

#### parameter

formal argument

formal parameter

<function or catch clause> object or reference declared as part of a function declaration or definition or in the catch clause of an exception handler that acquires a value on entry to the function or handler

1.3.15 [defns.parameter.macro]

parameter

formal argument

formal parameter

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

1.3.16 [defns.parameter.templ]

parameter

formal argument

formal parameter

<template> template-parameter

1.3.17 [defns.signature]

signature

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

## 1.3.18 [defns.signature.templ]

#### signature

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

1.3.19 [defns.signature.spec]

#### signature

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

### 1.3.20 [defns.signature.member]

#### signature

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

## 1.3.21 [defns.signature.member.templ]

#### signature

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

#### 1.3.22 [defns.signature.member.spec]

#### signature

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

 $[{\tt defns.static.type}]$ 

#### static type

type of an expression (3.9) resulting from analysis of the program without considering execution semantics [Note: The static type of an expression depends only on the form of the program in which the expression appears, and does not change while the program is executing. — end note]

1.3.24 [defns.undefined]

#### undefined behavior

behavior for which this International Standard imposes no requirements

[ Note: Undefined behavior may be expected when this International Standard omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed. —  $end\ note$ 

## 1.3.25 [defns.unspecified]

#### unspecified behavior

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

[Note: The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this International Standard.  $-end\ note$ ]

1.3.26 [defns.well.formed]

#### well-formed program

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

#### 1.4 Implementation compliance

#### [intro.compliance]

- <sup>1</sup> 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."
- <sup>2</sup> 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<sup>2</sup> 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.
- <sup>3</sup> 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.
- <sup>4</sup> For functions, function templates, objects, and values, the library Clauses specify declarations. Implementations shall supply definitions consistent with the descriptions in the library Clauses.
- <sup>5</sup> 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).
- 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).
- <sup>7</sup> Two kinds of implementations are defined: a hosted implementation and a freestanding implementation. For a hosted implementation, this International Standard defines the set of available libraries. A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation-defined set of libraries that includes certain language-support libraries (17.6.1.3).
- A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any well-formed program. Implementations are required to diagnose programs that use such extensions that are ill-formed according to this International Standard. Having done so, however, they can compile and execute such programs.
- Each implementation shall include documentation that identifies all conditionally-supported constructs that it does not support and defines all locale-specific characteristics.<sup>3</sup>

#### 1.5 Structure of this International Standard

[intro.structure]

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.

<sup>2) &</sup>quot;Correct execution" can include undefined behavior, depending on the data being processed; see 1.3 and 1.9.

<sup>3)</sup> This documentation also defines implementation-defined behavior; see 1.9.

<sup>2</sup> 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.

- 3 Annex B recommends lower bounds on the capacity of conforming implementations.
- <sup>4</sup> 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 [syntax]

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

{  $expression_{opt}$ }

indicates an optional expression enclosed in braces.

- <sup>2</sup> 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

[intro.memory]

- The fundamental storage unit in the C++ memory model is the byte. A byte is at least large enough to contain any member of the basic execution character set (2.3) and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.
- <sup>2</sup> [Note: The representation of types is described in 3.9. end note]
- A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. [Note: Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. —end note] Two or more threads of execution (1.10) can update and access separate memory locations without interfering with each other.
- <sup>4</sup> [Note: Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields of non-zero width. end note]
- <sup>5</sup> [Example: A structure declared as

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

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

#### 1.8 The C++ object model

[intro.object]

- The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object is a region of storage. [Note: A function is not an object, regardless of whether or not it occupies storage in the way that objects do. end note] An object is created by a definition (3.1), by a new-expression (5.3.4) or by the implementation (12.2) when needed. The properties of an object are determined when the object is created. An object can have a name (Clause 3). An object has a storage duration (3.7) which influences its lifetime (3.8). An object has a type (3.9). The term object type refers to the type with which the object is created. Some objects are polymorphic (10.3); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the expressions (Clause 5) used to access them.
- <sup>2</sup> 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*.
- <sup>3</sup> 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.
- <sup>4</sup> 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.
- <sup>6</sup> Unless an object is a bit-field or a base class subobject of zero size, the address of that object is the address of the first byte it occupies. Two objects that are not bit-fields may have the same address if one is a subobject of the other, or if at least one is a base class subobject of zero size and they are of different types; otherwise, they shall have distinct addresses.<sup>4</sup>

[ Example:

```
static const char test1 = 'x';
static const char test2 = 'x';
const bool b = &test1 != &test2;  // always true
— end example]
```

<sup>7</sup> [Note: C++ provides a variety of fundamental types and several ways of composing new types from existing types (3.9). — end note]

<sup>4)</sup> 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).

#### 1.9 Program execution

[intro.execution]

<sup>1</sup> 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.<sup>5</sup>

- <sup>2</sup> 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.<sup>6</sup> Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the "corresponding instance" below).
- <sup>3</sup> 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 attempting to modify a const object). [Note: This International Standard imposes no requirements on the behavior of programs that contain undefined behavior. —end note]
- A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).
- 6 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 during the execution of the signal handler, 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).
- 8 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.

<sup>5)</sup> 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.

<sup>6)</sup> This documentation also includes conditionally-supported constructs and locale-specific behavior. See 1.4.

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

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

```
int a, b;

/* ... */

a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

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

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

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

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

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

or

a = (a + (b + 32765)):
```

[Example:

{ } }

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

A full-expression is an expression that is not a subexpression of another expression. If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. A call to a destructor generated at the end of the lifetime of an object other than a temporary object is an implicit full-expression. Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression.

§ 1.9

// temporary is deleted at end of full-expression

<sup>7)</sup> Overloaded operators are never assumed to be associative or commutative.

— end example]

[Note: The evaluation of a full-expression can include the evaluation of subexpressions that are not lexically part of the full-expression. For example, subexpressions involved in evaluating default arguments (8.3.6) are considered to be created in the expression that calls the function, not the expression that defines the default argument. — end note

- Accessing an object designated by a volatile glvalue (3.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. *Evaluation* of an expression (or a sub-expression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access to a volatile object is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the volatile access may not have completed yet.
- 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.<sup>8</sup>.
- 15 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:

— end example]

When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. [Note: Value computations and side effects associated with different argument expressions are unsequenced. — end note] Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function. Several contexts in C++ cause evaluation of a function call,

<sup>8)</sup> 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.

<sup>9)</sup> In other words, function executions do not interleave with each other.

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

#### [intro.multithread]

- 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. Once a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of all of its threads. [Note: Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. —end note] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.
- <sup>2</sup> Implementations should ensure that all unblocked threads eventually make progress. [Note: Standard library functions may silently block on I/O or locks. Factors in the execution environment, including externally-imposed thread priorities, may prevent an implementation from making certain guarantees of forward progress. —end note]
- <sup>3</sup> 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]
- <sup>4</sup> Two expression evaluations *conflict* if one of them modifies a memory location (1.7) and the other one accesses or modifies the same memory location.
- The library defines a number of atomic operations (Clause 29) and operations on mutexes (Clause 30) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics. [Note: For example, a call that acquires a mutex will perform an acquire operation on the locations comprising the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on A. "Relaxed" atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. end note]
- 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

<sup>10)</sup> 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 objects. In general this will be impossible since different threads may observe modifications to different objects in inconsistent orders.  $-end\ note$ 

- A release sequence headed by a release operation A on an atomic object M is a maximal contiguous subsequence of side effects in the modification order of M, where the first operation is A, and every subsequent operation
  - is performed by the same thread that performed A, or
  - is an atomic read-modify-write operation.
- 8 Certain library calls synchronize with other library calls performed by another thread. For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store (29.3). [Note: Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. —end note] [Note: The specifications of the synchronization operations define when one reads the value written by another. For atomic objects, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition "reads the value written" by the last mutex release. —end note]
- <sup>9</sup> 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 intrathread. — end note]

- An evaluation A is dependency-ordered before an evaluation B if
  - A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or
  - for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

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

- An evaluation A inter-thread happens before an evaluation B if
  - A synchronizes with B, or
  - A is dependency-ordered before B, or
  - for some evaluation X
    - A synchronizes with X and X is sequenced before B, or
    - A is sequenced before X and X inter-thread happens before B, or

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

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

- <sup>12</sup> An evaluation A happens before an evaluation B if:
  - A is sequenced before B, or
  - A inter-thread happens before B.

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

- A visible side effect A on a scalar object or bit-field M with respect to a value computation B of M satisfies the conditions:
  - 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 below, 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 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. [Note: It can be shown that the visible sequence of side effects of a value computation is unique given the coherence requirements below. end note]
- If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M. [Note: This requirement is known as write-write coherence. end note]
- If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or the value stored by a side effect Y on M, where Y follows X in the modification order of M. [Note: This requirement is known as read-read coherence. end note]
- If a value computation A of an atomic object M happens before an operation B on M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M. [Note: This requirement is known as read-write coherence. end note]

If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M. [Note: This requirement is known as write-read coherence. — end note]

- 19 [Note: The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations. end note]
- [Note: The visible sequence of side effects 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 mutexes and memory\_order\_seq\_cst operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must perform an undefined operation. end note]
- [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]
- [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]
- <sup>24</sup> The implementation may assume that any thread will eventually do one of the following:
  - terminate,
  - make a call to a library I/O function,
  - access or modify a volatile object, or
  - perform a synchronization operation or an atomic operation.

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

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

#### 1.11 Acknowledgments

[intro.ack]

<sup>1</sup> The C++ programming language as described in this International Standard is based on the language as described in Chapter R (Reference Manual) of Stroustrup: *The C++ Programming Language* (second edition, Addison-Wesley Publishing Company, ISBN 0-201-53992-6, copyright ©1991 AT&T). That, in turn, is based

- on the C programming language as described in Appendix A of Kernighan and Ritchie: The C Programming Language (Prentice-Hall, 1978, ISBN 0-13-110163-3, copyright ©1978 AT&T).
- <sup>2</sup> Portions of the library Clauses of this International Standard are based on work by P.J. Plauger, which was published as *The Draft Standard C++ Library* (Prentice-Hall, ISBN 0-13-117003-1, copyright ©1995 P.J. Plauger).
- <sup>3</sup> POSIX® is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc.
- <sup>4</sup> All rights in these originals are reserved.

§ 1.11 15

## 2 Lexical conventions

[lex]

#### 2.1 Separate translation

[lex.separate]

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

[lex.phases]

- The precedence among the syntax rules of translation is specified by the following phases.<sup>11</sup>
  - 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 \uxxxxx notation), are handled equivalently except where this replacement is reverted in a raw string literal.)
  - 2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. 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. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is unspecified. The process of dividing a source file's characters into preprocessing tokens is context-dependent. [Example: see the handling of < within a #include preprocessing directive. end example]
  - 4. Preprocessing directives are executed, macro invocations are expanded, and \_Pragma unary operator expressions are executed. If a character sequence that matches the syntax of a universal-character-name

<sup>11)</sup> Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.

<sup>12)</sup> A partial preprocessing token would arise from a source file ending in the first portion of a multi-character token that requires a terminating sequence of characters, such as a *header-name* that is missing the closing " or >. A partial comment would arise from a source file ending with an unclosed /\* comment.

is produced by token concatenation (16.3.3), the behavior is undefined. A **#include** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

- 5. Each source character set member in a character literal or a string literal, as well as each escape sequence and universal-character-name in a character literal or a non-raw string literal, is converted to the corresponding member of the execution character set (2.14.3, 2.14.5); if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.<sup>13</sup>
- 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 [lex.charset]

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

- { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

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

hex-quad:

 $hexadecimal-digit\ hexadecimal-digit\ hexadecimal-digit\ hexadecimal-digit$ 

<sup>13)</sup> An implementation need not convert all non-corresponding source characters to the same execution character.

<sup>14)</sup> 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.

universal-character-name:

\u hex-quad

\U hex-quad hex-quad

The character designated by the universal-character-name \UNNNNNNN is that character whose character short name in ISO/IEC 10646 is NNNNNNNN; the character designated by the universal-character-name \uNNNN is that character whose character short name in ISO/IEC 10646 is 0000NNNN. If the hexadecimal value for a universal-character-name corresponds to a surrogate code point (in the range 0xD800–0xDFFF, inclusive), the program is ill-formed. Additionally, if the hexadecimal value for a universal-character-name outside 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. <sup>15</sup>

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

[lex.trigraph]

<sup>1</sup> 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.

Trigraph	Replacement	Trigraph	Replacement	Trigraph	Replacement
??=	#	??(	[	??<	{
??/	\	??)	]	??>	}
??'	^	??!		??-	~

Table 1 — Trigraph sequences

#### 2 [Example:

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

becomes

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

- end example]
- <sup>3</sup> No other trigraph sequence exists. Each ? that does not begin one of the trigraphs listed above is not changed.

<sup>15)</sup> A sequence of characters resembling a universal-character-name in an r-char-sequence (2.14.5) does not form a universal-character-name.

#### 2.5 Preprocessing tokens

[lex.pptoken]

```
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

<sup>1</sup> 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.

- 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.
- <sup>3</sup> 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. Between the initial and final double quote characters of the raw string, any transformations performed in phases 1 and 2 (trigraphs, universal-character-names, and line splicing) are reverted; this reversion shall apply before any d-char, r-char, or delimiting parenthesis is identified. The raw string literal is defined as the shortest sequence of characters that matches the raw-string pattern

```
encoding-prefix_{opt}R raw-string
```

- Otherwise, if the next three characters are <:: and the subsequent character is neither: nor >, the < is treated as a preprocessor token by itself and not as the first character of the alternative token <:.
- 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]
```

- <sup>4</sup> [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]
- <sup>5</sup> [Example: The program fragment x+++++y is parsed as x ++ ++ + y, which, if x and y have integral types, violates a constraint on increment operators, even though the parse x ++ ++ y might yield a correct expression. end example]

#### 2.6 Alternative tokens

[lex.digraph]

- Alternative token representations are provided for some operators and punctuators. 16
- <sup>2</sup> In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.<sup>17</sup> The set of alternative tokens is defined in Table 2.

Alternative	Primary	Alternative	Primary	Alternative	Primary
<%	{	and	&&	and_eq	&=
%>	}	bitor		or_eq	=
<:	[	or	П	xor_eq	^=
:>	]	xor	^	not	!
%:	#	compl	~	not_eq	!=
%:%:	##	bitand	&		

Table 2 — Alternative tokens

2.7 Tokens [lex.token]

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

There are five kinds of tokens: identifiers, keywords, literals, <sup>18</sup> 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 [lex.comment]

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 [lex.header]

§ 2.9 20

<sup>16)</sup> These include "digraphs" and additional reserved words. The term "digraph" (token consisting of two characters) is not perfectly descriptive, since one of the alternative preprocessing-tokens is %:%: and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren't lexical keywords are colloquially known as "digraphs".

<sup>17)</sup> Thus the "stringized" values (16.3.2) of [ and <: will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

<sup>18)</sup> Literals include strings and character and numeric literals.

```
q\text{-}char\text{-}sequence: q\text{-}char q\text{-}char\text{-}sequence\ q\text{-}char q\text{-}char: any member of the source character set except new-line and "
```

<sup>1</sup> 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.  $^{19}$ 

#### 2.10 Preprocessing numbers

[lex.ppnumber]

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

- <sup>1</sup> Preprocessing number tokens lexically include all integral literal tokens (2.14.2) and all floating literal tokens (2.14.4).
- 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 [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
```

An identifier is an arbitrarily long sequence of letters and digits. Each universal-character-name in an identifier shall designate a character whose encoding in ISO 10646 falls into one of the ranges specified in E.1. The initial element shall not be a universal-character-name designating a character whose encoding falls into one of the ranges specified in E.2. Upper- and lower-case letters are different. All characters are significant.<sup>20</sup>

§ 2.11 21

<sup>19)</sup> 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.

<sup>20)</sup> On systems in which linkers cannot accept extended characters, an encoding of the universal-character-name may be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the \u in a universal-character-name. Extended characters may produce a long external identifier, but C++ does not place a translation limit on significant characters for external identifiers. In C++, upper- and lower-case letters are considered different for all identifiers, including external identifiers.

The identifiers in Table 3 have a special meaning when appearing in a certain context. When referred to in the grammar, these identifiers are used explicitly rather than using the *identifier* grammar production. any ambiguity as to whether a given identifier has a special meaning is resolved to interpret the token as a regular identifier.

Table 3 — Identifiers with special meaning

override final

In addition, some identifiers are reserved for use by C++ implementations and standard libraries (17.6.4.3.2) and shall not be used otherwise; no diagnostic is required.

#### 2.12 Keywords [lex.key]

The identifiers shown in Table 4 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 4 — Keywords

		•	
lignas	continue	friend	reg

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

Furthermore, the alternative representations shown in Table 5 for certain operators and punctuators (2.6) are reserved and shall not be used otherwise:

Table 5 — Alternative representations

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	xor	xor_eq	

#### Operators and punctuators

[lex.operators]

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

```
preprocessing \hbox{-} op \hbox{-} or \hbox{-} punc:
       {
               }
                             Ε
                                                                     ##
                                          ٦
                                          %>
                                                       %:
                                                                     %:%:
       <:
                            <%
               :>
                            ?
               delete
       new
                                                       %
               &=
       <=
                            &&
                                          П
                                                       ++
               {\tt and}\_{\tt eq}
                            bitand
                                          bitor
                                                       compl
       and
                                                                     not
                                                                                  not_eq
                            xor
                                          xor_eq
       or
               or_eq
```

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

# 2.14 Literals

[lex.literal]

#### 2.14.1 Kinds of literals

[lex.literal.kinds]

There are several kinds of literals.<sup>21</sup> literal:

2.14.2

integer-literal

 $character\mbox{-}literal$ 

floating-literal

string-literal

boolean-literal

pointer-literal

## user-defined-literalInteger literals

[lex.icon]

```
integer\mbox{-}literal:
```

decimal-literal integer-suffix $_{opt}$ 

octal-literal integer-suffix<sub>opt</sub>

hexadecimal-literal integer-suffix $_{opt}$ 

 $decimal\mbox{-}literal\mbox{:}$ 

nonzero-digit

 $decimal\text{-}literal\ digit$ 

 $octal ext{-}literal:$ 

octal-literal octal-digit

hexadecimal-literal:

0x hexadecimal-digit

OX 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

ABCDEF

integer-suffix:

unsigned-suffix long-suffix $_{opt}$ 

unsigned-suffix long-long-suffix $_{opt}$ 

long-suffix unsigned-suffix $_{opt}$ 

long-long-suffix unsigned- $suffix_{opt}$ 

<sup>21)</sup> The term "literal" generally designates, in this International Standard, those tokens that are called "constants" in ISO C.

 $\begin{array}{c} unsigned\text{-}suffix: \text{ one of} \\ u \ U \\ long\text{-}suffix: \text{ one of} \\ 1 \ L \\ long\text{-}long\text{-}suffix: \text{ one of} \end{array}$ 

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. An octal 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]

<sup>2</sup> The type of an integer literal is the first of the corresponding list in Table 6 in which its value can be represented.

Suffix	Decimal constants	Octal or hexadecimal constant			
none	int	int			
	long int	unsigned int			
	long long int	long int			
		unsigned long int			
		long long int			
		unsigned long long int			
u or U	unsigned int	unsigned int			
	unsigned long int	unsigned long int			
	unsigned long long int	unsigned long long int			
1 or L	long int	long int			
	long long int	unsigned long int			
		long long int			
		unsigned long long int			
Both u or U	unsigned long int	unsigned long int			
and 1 or L	unsigned long long int	unsigned long long int			
11 or LL	long long int	long long int			
		unsigned long long int			
Both u or U	unsigned long long int	unsigned long long int			
and 11 or LL					

Table 6 — Types of integer constants

If an integer literal cannot be represented by any type in its list and an extended integer type (3.9.1) can represent its value, it may have that extended integer type. If all of the types in the list for the literal are signed, the extended integer type shall be signed. If all of the types in the list for the literal are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. A program is ill-formed if one of its translation units contains an

<sup>22)</sup> The digits 8 and 9 are not octal digits.

integer literal that cannot be represented by any of the allowed types.

#### 2.14.3 Character literals

[lex.ccon]

```
character\hbox{-}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\mbox{-}sequence
      universal-character-name
escape-sequence:
      simple-escape-sequence
      octal-escape-sequence
      hexadecimal-escape-sequence
simple-escape-sequence: one of
      \'
            \"
                 \?
                        11
            \b
      \a
                \f
                       \n
                                         ١v
                             \r
                                    \t
octal\mbox{-}escape\mbox{-}sequence:
      \ octal-digit
      \ octal-digit octal-digit
      \ \ \ \  octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:
      \x hexadecimal-digit
```

hexadecimal-escape-sequence hexadecimal-digit

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

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.<sup>23</sup> 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.

<sup>23)</sup> They are intended for character sets where a character does not fit into a single byte.

<sup>3</sup> Certain nongraphic characters, the single quote ', the double quote ", the question mark ?, <sup>24</sup> and the backslash \, can be represented according to Table 7. The double quote " and the question mark ?, can be represented as themselves or by the escape sequences \" and \? respectively, but the single quote ' and the backslash \ shall be represented by the escape sequences \' and \\ respectively. Escape sequences in which the character following the backslash is not listed in Table 7 are conditionally-supported, with implementation-defined semantics. An escape sequence specifies a single character.

new-line	NL(LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	$\operatorname{CR}$	\r
form feed	FF	\f
alert	$\operatorname{BEL}$	\a
backslash	\	\\
question mark	?	\?
single quote	,	\',
double quote	II	\"
octal number	000	\000
hex number	hhh	$\xhhh$

Table 7 — Escape sequences

- The escape \ooo consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \xhhh consists of the backslash followed by x followed by one or more hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of digits in a hexadecimal sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character literal is implementation-defined if it falls outside of the implementation-defined range defined for char (for literals with no prefix), char16\_t (for literals prefixed by 'u'), char32\_t (for literals prefixed by 'U'), or wchar\_t (for literals prefixed by 'L').
- A universal-character-name is translated to the encoding, in the 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

[lex.fcon]

```
floating-literal: \\ fractional-constant \ exponent-part_{opt} \ floating-suffix_{opt} \\ digit-sequence \ exponent-part \ floating-suffix_{opt} \\ fractional-constant: \\ digit-sequence_{opt}. \ digit-sequence \\ digit-sequence . \\ exponent-part: \\ e \ sign_{opt} \ digit-sequence \\ E \ sign_{opt} \ digit-sequence \\ E \ sign_{opt} \ digit-sequence \\ \end{cases}
```

<sup>24)</sup> Using an escape sequence for a question mark can avoid accidentally creating a trigraph.

A floating literal consists of an integer part, a decimal point, a fraction part, an e or E, an optionally signed integer exponent, and an optional type suffix. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (not both) can be omitted; either the decimal point or the letter e (or E) and the exponent (not both) can be omitted. The integer part, the optional decimal point and the optional fraction part form the significant part of the floating literal. The exponent, if present, indicates the power of 10 by which the significant part is to be scaled. If the scaled value is in the range of representable values for its type, the result is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner. The type of a floating literal is double unless explicitly specified by a suffix. The suffixes f and F specify float, the suffixes 1 and L specify long double. If the scaled value is not in the range of representable values for its type, the program is ill-formed.

#### 2.14.5 String literals

[lex.string]

```
string-literal:
      encoding-prefix_{opt}" s-char-sequence_{opt}"
      encoding-prefix<sub>opt</sub>R raw-string
encoding	ext{-}prefix:
      u8
      u
      U
      L
s-char-sequence:
      s-char
      s\text{-}char\text{-}sequence\ s\text{-}char
s-char:
      any member of the source character set except
             the double-quote ", backslash \, or new-line character
      escape\text{-}sequence
      universal-character-name
       " d-char-sequence<sub>opt</sub> ( r-char-sequence<sub>opt</sub> ) d-char-sequence<sub>opt</sub> "
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.
```

A string literal is a sequence of characters (as defined in 2.14.3) surrounded by double quotes, optionally prefixed by R, u8, u8, u, uR, U, UR, L, or LR, as in "...", R"(...)", u8"...", u8"\*\*(...)\*\*", u"...", uR"\*\*(...)\*", UR"zzz(...)zzz", 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.
- 1 [Note: The characters '(' and ')' are permitted in a raw-string. Thus, R"delimiter((a|b))delimiter"
  is equivalent to "(a|b)". end note]
- <sup>4</sup> [Note: A source-file new-line in a raw string literal results in a new-line in the resulting execution string-literal. Assuming no whitespace at the beginning of lines in the following example, the assert will succeed:

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

— end note]
[Example: The raw string

R"a(
)\
a"
)a"

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

R"(??)"

is equivalent to "\?\?". The raw string

R"#(
)??="
)#"
```

- is equivalent to "\n)\?\?=\"\n".  $-end\ example$ ]
- <sup>6</sup> 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.
- <sup>7</sup> 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.
- <sup>8</sup> 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).
- <sup>9</sup> A string literal that begins with u, such as u"asdf", is a char16\_t string literal. A char16\_t string literal has type "array of n const char16\_t", where n is the size of the string as defined below; it has static storage duration and is initialized with the given characters. A single c-char may produce more than one char16\_t character in the form of surrogate pairs.
- A string literal that begins with U, such as U"asdf", is a char32\_t string literal. A char32\_t string literal has type "array of n const char32\_t", where n is the size of the string as defined below; it has static storage duration and is initialized with the given characters.
- A string literal that begins with L, such as L"asdf", is a wide string literal. A wide string literal has type "array of n const wchar\_t", where n is the size of the string as defined below; it has static storage duration and is initialized with the given characters.
- Whether all string literals are distinct (that is, are stored in nonoverlapping objects) is implementationdefined. The effect of attempting to modify a string literal is undefined.
- <sup>13</sup> 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 literal's initial rawness has no effect on the interpretation or well-formedness of the concatenation. -end note Table 8 has some examples of valid concatenations.

Table 8 — String literal concatenations

Source		Means	Source		Means	Source		Means
u"a"	u"b"	u"ab"	U"a"	U"b"	U"ab"	L"a"	L"b"	L"ab"
u"a"	"b"	u"ab"	U"a"	"b"	U"ab"	L"a"	"b"	L"ab"
"a"	u"b"	u"ab"	"a"	U"b"	U"ab"	"a"	L"b"	L"ab"

Characters in concatenated strings are kept distinct. [ Example:

"\xA" "B"

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

- After any necessary concatenation, in translation phase 7 (2.2), '\0' is appended to every string literal so that programs that scan a string can find its end.
- Escape sequences and universal-character-names in non-raw string literals have the same meaning as in character literals (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

[lex.bool]

 $boolean\hbox{-}literal:$ 

false

true

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

#### 2.14.7 Pointer literals

[lex.nullptr]

pointer-literal:

nullptr

The pointer literal is the keyword nullptr. It is a prvalue of type std::nullptr\_t. [Note: std::nullptr\_t is a distinct type that is neither a pointer type nor a pointer to member type; rather, a prvalue of this type is

a null pointer constant and can be converted to a null pointer value or null member pointer value. See 4.10 and 4.11. — end note

## 2.14.8 User-defined literals

[lex.ext]

```
user-defined-literal:
       user\text{-}defined\text{-}integer\text{-}literal
       user-defined-floating-literal
       user-defined-string-literal
       user\text{-}defined\text{-}character\text{-}literal
user-defined-integer-literal:
       decimal-literal ud-suffix
       octal-literal ud-suffix
       hexadecimal-literal ud-suffix
user-defined-floating-literal:
       fractional\text{-}constant\ exponent\text{-}part_{opt}\ ud\text{-}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
```

- If a token matches both user-defined-literal and another literal kind, it is treated as the latter. [Example: 123\_km is a user-defined-literal, but 12LL is an integer-literal. end example] The syntactic non-terminal preceding the ud-suffix in a user-defined-literal is taken to be the longest sequence of characters that could match that non-terminal.
- 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.
- 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 < c_1', c_2', \ldots c_k' > ()
```

where n is the source character sequence  $c_1c_2...c_k$ . [Note: The sequence  $c_1c_2...c_k$  can only contain characters from the basic source character set. — end note]

<sup>4</sup> 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

```
operator "" X < c_1', c_2', \ldots c_k' > ()
```

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

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

```
operator "" X (str, len)
```

<sup>6</sup> If L is a user-defined-character-literal, let ch be the literal without its ud-suffix. S shall contain a literal operator (13.5.8) whose only parameter has the type of ch and the literal L is treated as a call of the form

```
operator "" X (ch)
```

<sup>7</sup> [Example:

— 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 [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]

Some *identifiers* appearing as *ud-suffixes* are reserved for future standardization (17.6.4.3.5). A program containing such a *ud-suffix* is ill-formed, no diagnostic required.

# 3 Basic concepts

[basic]

[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]

- <sup>2</sup> [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]
- <sup>3</sup> An *entity* is a value, object, reference, function, enumerator, type, class member, template, template specialization, namespace, parameter pack, or this.
- <sup>4</sup> 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).
- <sup>5</sup> 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).
- <sup>6</sup> 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.
- <sup>7</sup> 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.
- 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]

- <sup>1</sup> A declaration (Clause 7) may introduce one or more names into a translation unit or redeclare names introduced by previous declarations. If so, the declaration specifies the interpretation and attributes of these names. A declaration may also have effects including:
  - a static assertion (Clause 7),
  - controlling template instantiation (14.7.2),
  - use of attributes (Clause 7), and
  - nothing (in the case of an *empty-declaration*).

§ 3.1 32

A declaration is a definition unless it declares a function without specifying the function's body (8.4), it contains the extern specifier (7.1.1) or a linkage-specification<sup>25</sup> (7.5) and neither an initializer nor a function-body, it declares a static data member in a class definition (9.2, 9.4), it is a class name declaration (9.1), it is an opaque-enum-declaration (7.2), it is a template-parameter (14.1), it is a parameter-declaration (8.3.5) in a function declarator that is not the declarator of a function-definition, or it is a typedef declaration (7.1.3), an alias-declaration (7.1.3), a using-declaration (7.3.3), a static\_assert-declaration (Clause 7), an attribute-declaration (Clause 7), an empty-declaration (Clause 7), or a using-directive (7.3.4).

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

```
int a;
                                     // defines a
  extern const int c = 1;
                                     // defines c
                                     // defines f and defines x
  int f(int x) { return x+a; }
                                     // defines S, S::a, and S::b
  struct S { int a; int b; };
  struct X {
                                     // defines X
                                     // defines non-static data member x
    int x;
                                     // declares static data member y
    static int y;
    X(): x(0) \{ \}
                                     // defines a constructor of X
 };
  int X::y = 1;
                                     // defines X::y
                                     // defines up and down
  enum { up, down };
 namespace N { int d; }
                                     // defines N and N::d
                                     // defines N1
 namespace N1 = N;
 X anX;
                                     // defines anX
whereas these are just declarations:
                                     // declares a
  extern int a;
                                     // declares c
  extern const int c;
                                     // declares f
  int f(int);
  struct S;
                                     // declares S
  typedef int Int;
                                     // declares Int
  extern X anotherX;
                                     // declares anotherX
                                     // declares d
 using N::d;
— end example]
```

[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

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

§ 3.1 33

```
C() : s() { }
C(const C& x): s(x.s) { }
C(C&& x): s(static_cast<std::string&&>(x.s)) { }
    //: s(std::move(x.s)) { }
C& operator=(const C& x) { s = x.s; return *this; }
C& operator=(C&& x) { s = static_cast<std::string&&>(x.s); return *this; }
    // { s = std::move(x.s); return *this; }
    *C() { }
};
— end example]
```

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

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]

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

- An expression is potentially evaluated unless it is an unevaluated operand (Clause 5) or a subexpression thereof. A variable whose name appears as a potentially-evaluated expression is odr-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 odr-used if it appears as a potentially-evaluated expression (including as the result of the implicit transformation in the body of a non-static member function (9.3.1)). A virtual member function is odr-used if it is not pure. A non-overloaded function whose name appears as a potentially-evaluated expression or a member of a set of candidate functions, if selected by overload resolution when referred to from a potentially-evaluated expression, is odr-used, unless it is a pure virtual function and its name is not explicitly qualified. [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 odr-used even if the call is actually elided by the implementation. — end note An allocation or deallocation function for a class is odr-used by a new expression appearing in a potentially-evaluated expression as specified in 5.3.4 and 12.5. A deallocation function for a class is odr-used by a delete expression appearing in a potentially-evaluated expression as specified in 5.3.5 and 12.5. A non-placement allocation or deallocation function for a class is odr-used by the definition of a constructor of that class. A non-placement deallocation function for a class is odr-used by the definition of the destructor of that class, or by being selected by the lookup at the point of definition of a virtual destructor (12.4).<sup>26</sup> A copy-assignment function for a class is odr-used by an implicitlydefined copy-assignment function for another class as specified in 12.8. A move-assignment function for a class is odr-used by an implicitly-defined move-assignment function for another class as specified in 12.8. A default constructor for a class is odr-used by default initialization or value initialization as specified in 8.5. A constructor for a class is odr-used as specified in 8.5. A destructor for a class is odr-used as specified in 12.4.
- <sup>3</sup> Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program; no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 12.1, 12.4 and 12.8). An inline function shall be defined in every translation unit in which it is odr-used.
- <sup>4</sup> 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
```

§ 3.2

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

```
X* x2; // use X in pointer formation
```

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

- an object of type T is defined (3.1), or
- a non-static class data member of type T is declared (9.2), or
- T is used as the object type or array element type in a new-expression (5.3.4), or
- an lvalue-to-rvalue conversion is applied to a 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 void\*, is converted to the type pointer to T or reference to T using an implicit conversion (Clause 4), a dynamic\_cast (5.2.7) or a static\_cast (5.2.9), or
- a class member access operator is applied to an expression of type T (5.2.5), or
- the typeid operator (5.2.8) or the size of 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 (Clause 10), or
- an lvalue of type T is assigned to (5.17), or
- the type T is the subject of an alignof expression (5.3.6), or
- an exception-declaration has type T, reference to T, or pointer to T (15.3).
- -end note
- There can be more than one definition of a class type (Clause 9), enumeration type (7.2), inline function with external linkage (7.1.2), class template (Clause 14), non-static function template (14.5.6), static data member of a class template (14.5.1.3), member function of a class template (14.5.1.1), or template specialization for which some template parameters are not specified (14.7, 14.5.5) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements. Given such an entity named D defined in more than one translation unit, then
  - each definition of D shall consist of the same sequence of tokens; and
  - in each definition of D, corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of D, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a const object with internal or no linkage if the object has the same literal type in all definitions of D, and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of D; and
  - in each definition of D, corresponding entities shall have the same language linkage; and
  - in each definition of D, the overloaded operators referred to, the implicit calls to conversion functions, constructors, operator new functions and operator delete functions, shall refer to the same function, or to a function defined within the definition of D; and

§ 3.2 35

— in each definition of D, a default argument used by an (implicit or explicit) function call is treated as if its token sequence were present in the definition of D; that is, the default argument is subject to the three requirements described above (and, if the default argument has sub-expressions with default arguments, this requirement applies recursively).<sup>27</sup>

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

```
//translation unit 1:
 struct X {
   X(int);
   X(int, int);
 };
 X::X(int = 0) { }
 class D: public X { };
 D d2;
                                    // X(int) called by D()
 //translation unit 2:
 struct X {
   X(int);
   X(int, int);
 };
 X::X(int = 0, int = 0) { }
 class D: public X { };
                                    // X(int, int) called by D();
                                    // D() 's implicit definition
                                     // violates the ODR
— end example
```

If D is a template and is defined in more than one translation unit, then the preceding requirements shall apply both to names from the template's enclosing scope used in the template definition (14.6.3), and also to dependent names at the point of instantiation (14.6.2). If the definitions of D satisfy all these requirements, then the 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 [basic.scope]

## 3.3.1 Declarative regions and scopes

## [basic.scope.declarative]

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.

<sup>2</sup> [Example: in

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

27) 8.3.6 describes how default argument names are looked up.

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

- <sup>3</sup> The names declared by a declaration are introduced into the scope in which the declaration occurs, except that the presence of a **friend** specifier (11.3), certain uses of the *elaborated-type-specifier* (7.1.6.3), and *using-directives* (7.3.4) alter this general behavior.
- 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 (11.3) may introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to that region. Local extern declarations (3.5) may introduce a name into the declarative region where the declaration appears and also introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to both regions. — end note]

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

#### 3.3.2 Point of declaration

[basic.scope.pdecl]

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:

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

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

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

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

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

- The point of declaration for a class or class template first declared by a class-specifier is immediately after the identifier or simple-template-id (if any) in its class-head (Clause 9). The point of declaration for an enumeration is immediately after the identifier (if any) in either its enum-specifier (7.2) or its first opaque-enum-declaration (7.2), whichever comes first. The point of declaration of an alias or alias template immediately follows the type-id to which the alias refers.
- <sup>4</sup> The point of declaration for an enumerator is immediately after its enumerator-definition. Example:

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

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

<sup>5</sup> 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,

<sup>6</sup> The point of declaration of a class first declared in an elaborated-type-specifier is as follows:

— for a declaration of the form

 $class-key \ attribute-specifier-seq_{opt} \ identifier$ ;

the identifier is declared to be a class-name in the scope that contains the declaration, otherwise

— for an elaborated-type-specifier of the form

class-key identifier

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

- <sup>7</sup> The point of declaration for an *injected-class-name* (Clause 9) is immediately following the opening brace of the class definition.
- The point of declaration for a function-local predefined variable (8.4) is immediately before the function-body of a function definition.
- The point of declaration for a template parameter is immediately after its complete template-parameter. [Example:

— end example]

- [Note: Friend declarations refer to functions or classes that are members of the nearest enclosing namespace, but they do not introduce new names into that namespace (7.3.1.2). Function declarations at block scope and variable declarations with the extern specifier at block scope refer to declarations that are members of an enclosing namespace, but they do not introduce new names into that scope. end note]
- 11 [Note: For point of instantiation of a template, see 14.6.4.1.—end note]

#### 3.3.3 Block scope

## [basic.scope.local]

- A name declared in a block (6.3) is local to that block; it has *block scope*. Its potential scope begins at its point of declaration (3.3.2) and ends at the end of its block. A variable declared at block scope is a *local variable*.
- The potential scope of a function parameter name (including one appearing in a lambda-declarator) or of a function-local predefined variable in a function definition (8.4) begins at its point of declaration. If the function has a function-try-block the potential scope of a parameter or of a function-local predefined variable ends at the end of the last associated handler, otherwise it ends at the end of the outermost block of the function definition. A parameter name shall not be redeclared in the outermost block of the function definition nor in the outermost block of any handler associated with a function-try-block.
- <sup>3</sup> The name declared in an *exception-declaration* is local to the *handler* and shall not be redeclared in the outermost block of the *handler*.

<sup>4</sup> Names declared in the *for-init-statement*, the *for-range-declaration*, and in the *condition* of if, while, for, and switch statements are local to the if, while, for, or switch statement (including the controlled statement), and shall not be redeclared in a subsequent condition of that statement nor in the outermost block (or, for the if statement, any of the outermost blocks) of the controlled statement; see 6.4.

#### 3.3.4 Function prototype scope

## [basic.scope.proto]

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

## 3.3.5 Function scope

[basic.funscope]

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

#### 3.3.6 Namespace scope

## [basic.scope.namespace]

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-definitions 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:

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

- end example]
- 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.
- <sup>3</sup> 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 name*.

## 3.3.7 Class scope

[basic.scope.class]

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

- 1) The potential scope of a name declared in a class consists not only of the declarative region following the name's point of declaration, but also of all function bodies, *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 {
                                        // error: T refers to ::T
   Ta;
                                        // but when reevaluated is Y::T
   typedef long T;
     b;
 };
 typedef int I;
 class D {
   typedef I I;
                                        // error, even though no reordering involved
 };
— end example]
```

- <sup>2</sup> 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

[basic.scope.enum]

<sup>1</sup> 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

[basic.scope.temp]

- <sup>1</sup> The declarative region of the name of a template parameter of a template template-parameter is the smallest template-parameter-list in which the name was introduced.
- The declarative region of the name of a template parameter of a template is the smallest template-declaration in which the name was introduced. Only template parameter names belong to this declarative region; any other kind of name introduced by the declaration of a template-declaration is instead introduced into the same declarative region where it would be introduced as a result of a non-template declaration of the same name. [Example:

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

The potential scope of a template parameter name begins at its point of declaration (3.3.2) and ends at the end of its declarative region. [Note: This implies that a template-parameter can be used in the declaration of subsequent template-parameters and their default arguments but cannot be used in preceding template-parameters or their default arguments. For example,

```
template<class T, T* p, class U = T> class X { /* \dots */ }; 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,

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

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.  $-end\ note$ 

<sup>4</sup> 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:

```
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. —  $end\ example$ ] —  $end\ note$ ]

<sup>5</sup> [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]

- A name can be hidden by an explicit declaration of that same name in a nested declarative region or derived class (10.2).
- <sup>2</sup> 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.
- 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.
- <sup>4</sup> 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).
- <sup>5</sup> If a name is in scope and is not hidden it is said to be *visible*.

#### 3.4 Name lookup

[basic.lookup]

- The name lookup rules apply uniformly to all names (including typedef-names (7.1.3), namespace-names (7.3), and class-names (9.1)) wherever the grammar allows such names in the context discussed by a particular rule. Name lookup associates the use of a name with a declaration (3.1) of that name. Name lookup shall find an unambiguous declaration for the name (see 10.2). Name lookup may associate more than one declaration with a name if it finds the name to be a function name; the declarations are said to form a set of overloaded functions (13.1). Overload resolution (13.3) takes place after name lookup has succeeded. The access rules (Clause 11) are considered only once name lookup and function overload resolution (if applicable) have succeeded. Only after name lookup, function overload resolution (if applicable) and access checking have succeeded are the attributes introduced by the name's declaration used further in expression processing (Clause 5).
- <sup>2</sup> 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.
- <sup>3</sup> 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.
- <sup>4</sup> [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.
- <sup>2</sup> 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,

```
typedef int f;
namespace N {
```

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

- <sup>4</sup> A name used in global scope, outside of any function, class or user-declared namespace, shall be declared before its use in global scope.
- <sup>5</sup> 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<sup>28</sup> that is a member of namespace N (where, only for the purpose of exposition, N could represent the global scope) shall be declared before its use in the block in which it is used or in one of its enclosing blocks (6.3) or, shall be declared before its use in namespace N or, if N is a nested namespace, shall be declared before its use in one of N's enclosing namespaces. [Example:

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

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

- end example]
- A name used in the definition of a class X outside of a member function body or nested class definition<sup>29</sup> 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

<sup>28)</sup> This refers to unqualified names that occur, for instance, in a type or default argument in the parameter-declaration-clause or used in the function body.

<sup>29)</sup> 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.

<sup>30)</sup> 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).

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

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

- end example] [Note: When looking for a prior declaration of a class or function introduced by a friend declaration, scopes outside of the innermost enclosing namespace scope are not considered; see 7.3.1.2. end note] [Note: 3.3.7 further describes the restrictions on the use of names in a class definition. 9.7 further describes the restrictions on the use of names in local class definitions. 9.8 further describes the restrictions on the use of names in local class definitions. -end note]
- A name used in the definition of a member function (9.3) of class X following the function's declarator-id<sup>31</sup> 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), <sup>32</sup> 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:

<sup>31)</sup> That is, an unqualified name that occurs, for instance, in a type or default argument in the parameter-declaration-clause or in the function body.

<sup>32)</sup> 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.

```
class B { };
namespace M {
  namespace N {
    class X : public B {
      void f();
    };
  }
}
void M::N::X::f() {
  i = 16;
}
// The following scopes are searched for a declaration of i:
// 1) outermost block scope of M::N::X::f, before the use of i
// 2) scope of class M::N::X
// 3) scope of M::N::X's base class B
// 4) scope of namespace M::N
// 5) scope of namespace M
// 6) global scope, before the definition of M::N::X::f
```

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

- Name lookup for a name used in the definition of a friend function (11.3) defined inline in the class granting friendship shall proceed as described for lookup in member function definitions. If the friend function is not defined in the class granting friendship, name lookup in the friend function definition shall proceed as described for lookup in namespace member function definitions.
- In a friend declaration naming a member function, a name used in the function declarator and not part of a template-argument in the declarator-id is first looked up in the scope of the member function's class (10.2). If it is not found, or if the name is part of a template-argument in the declarator-id, the look up is as described for unqualified names in the definition of the class granting friendship. [Example:

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

- end example

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

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

<sup>14</sup> 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:

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

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

#### 3.4.2 Argument-dependent name lookup

## [basic.lookup.argdep]

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

- end example]
- For each argument type T in the function call, there is a set of zero or more associated namespaces and a set of zero or more associated classes to be considered. The sets of namespaces and classes is determined entirely by the types of the function arguments (and the namespace of any template template argument). Typedef names and *using-declarations* used to specify the types do not contribute to this set. The sets of namespaces and classes are determined in the following way:
  - If T is a fundamental type, its associated sets of namespaces and classes are both empty.
  - If T is a class type (including unions), its associated classes are: the class itself; the class of which it is a member, if any; and its direct and indirect base classes. Its associated namespaces are the namespaces of which its associated classes are members. Furthermore, if T is a class template specialization, its associated namespaces and classes also include: the namespaces and classes associated with the

types of the template arguments provided for template type parameters (excluding template template parameters); the namespaces of which any template template arguments are members; and the classes of which any member templates used as template template arguments are members. [Note: Non-type template arguments do not contribute to the set of associated namespaces. — $end\ note$ ]

- If T is an enumeration type, its associated namespace is the namespace in which it is defined. If it is class member, its associated class is the member's class; else it has no associated class.
- If T is a pointer to U or an array of U, its associated namespaces and classes are those associated with
   U.
- If T is a function type, its associated namespaces and classes are those associated with the function parameter types and those associated with the return type.
- If T is a pointer to a member function of a class X, its associated namespaces and classes are those associated with the function parameter types and return type, together with those associated with X.
- If T is a pointer to a data member of class X, its associated namespaces and classes are those associated with the member type together with those associated with X.

If an associated namespace is an inline namespace (7.3.1), its enclosing namespace is also included in the set. If an associated namespace directly contains inline namespaces, those inline namespaces are also included in the set. In addition, if the argument is the name or address of a set of overloaded functions and/or function templates, its associated classes and namespaces are the union of those associated with each of the members of the set, i.e., the classes and namespaces associated with its parameter types and return type. Additionally, if the aforementioned set of overloaded functions is named with a template-id, its associated classes and namespaces also include those of its type template-arguments and its template template-arguments.

- <sup>3</sup> 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:

- end example]
- 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 (11.3).
  - All names except those of (possibly overloaded) functions and function templates are ignored.

#### 3.4.3 Qualified name lookup

## [basic.lookup.qual]

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:

- end example]
- <sup>2</sup> [Note: Multiply qualified names, such as N1::N2::N3::n, can be used to refer to members of nested classes (9.7) or members of nested namespaces. end note]
- In a declaration in which the *declarator-id* is a *qualified-id*, names used before the *qualified-id* being declared are looked up in the defining namespace scope; names following the *qualified-id* are looked up in the scope of the member's class or namespace. [Example:

- -end example
- A name prefixed by the unary scope operator :: (5.1) is looked up in global scope, in the translation unit where it is used. The name shall be declared in global namespace scope or shall be a name whose declaration is visible in global scope because of a *using-directive* (3.4.3.2). The use of :: allows a global name to be referred to even if its identifier has been hidden (3.3.10).
- <sup>5</sup> A name prefixed by a nested-name-specifier that nominates an enumeration type shall represent an enumerator of that enumeration.
- <sup>6</sup> 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:

```
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;
                      // I is looked up in the scope of C
p->C::I::~I();
q->I1::~I2();
                      // I2 is looked up in the scope of
                      // the postfix-expression
struct A {
  ~A();
};
typedef A AB;
int main() {
  AB *p;
  p->AB::~AB();
                      // explicitly calls the destructor for A
```

nested-name-specifier $_{opt}$  class-name ::  $\sim$  class-name

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

#### 3.4.3.1 Class members

[class.qual]

- If the nested-name-specifier of a qualified-id nominates a class, the name specified after the nested-name-specifier is looked up in the scope of the class (10.2), except for the cases listed below. The name shall represent one or more members of that class or of one of its base classes (Clause 10). [Note: A class member can be referred to using a qualified-id at any point in its potential scope (3.3.7). 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 in the same manner as a conversion-type-id in a class member access (see 3.4.5);
  - the names in a template-argument of a template-id are looked up in the context in which the entire postfix-expression occurs.
  - the lookup for a name specified in a *using-declaration* (7.3.3) also finds class or enumeration names hidden within the same scope (3.3.10).
- <sup>2</sup> 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:

§ 3.4.3.1 49

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

#### 3.4.3.2 Namespace members

[namespace.qual]

- 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. If a qualified-id starts with ::, the name after the :: is looked up in the global namespace. In either case, the names in a template-argument of a template-id are looked up in the context in which the entire postfix-expression occurs.
- 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.
- 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:

```
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;
```

§ 3.4.3.2

```
void g();
  void h()
   {
                          // g is declared directly in AB,
     AB::g();
                          // therefore S is { AB::g() } and AB::g() is chosen
                          // f is not declared directly in AB so the rules are
     AB::f(1);
                          // 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
                          // i is not declared directly in AB so the rules are
     AB::i++;
                          // applied recursively to A and B,
                          // S is { A::i , B::i } so the use is ambiguous
                          // and the program is ill-formed
                          // h is not declared directly in AB and
     AB::h(16.8);
                          // not declared directly in A or B so the rules are
                          // applied recursively to Y and Z,
                          // S is { Y::h(int), Z::h(double) } and overload
                          // resolution chooses Z::h(double)
   }
The same declaration found more than once is not an ambiguity (because it is still a unique declaration).
For example:
  namespace A {
     int a;
  }
  {\tt namespace} \ {\tt B} \ \{
     using namespace A;
  namespace C {
     using namespace A;
  namespace BC {
     using namespace B;
```

§ 3.4.3.2 51

// OK: S is { A::a, A::a }

using namespace C;

void f()
{

BC::a++;

namespace D {

<sup>5</sup> Because each referenced namespace is searched at most once, the following is well-defined:

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

— end example]

Ouring 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:

```
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 )
```

§ 3.4.3.2 52

-end example

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

 $nested\text{-}name\text{-}specifier\ unqualified\text{-}id$ 

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);
  }
  using namespace B;
}
void A::f1(int){ } // ill-formed, f1 is not a member of A
```

— end example] However, in such namespace member declarations, the nested-name-specifier may rely on using-directives to implicitly provide the initial part of the nested-name-specifier. [Example:

```
namespace A {
  namespace B {
    void f1(int);
  }
}

namespace C {
  namespace D {
    void f1(int);
  }
}

using namespace A;
using namespace C::D;
void B::f1(int){ } // OK, defines A::B::f1(int)

— end example]
```

## 3.4.4 Elaborated type specifiers

[basic.lookup.elab]

- 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).
- <sup>2</sup> If the *elaborated-type-specifier* has no *nested-name-specifier*, and unless the *elaborated-type-specifier* appears in a declaration with the following form:

 $class-key \ attribute-specifier-seq_{opt} \ identifier$ ;

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

class-key attribute-specifier-seq<sub>opt</sub> identifier;

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

If the elaborated-type-specifier has a nested-name-specifier, qualified name lookup is performed, as described in 3.4.3, but ignoring any non-type names that have been declared. If the name lookup does not find a previously declared type-name, the elaborated-type-specifier is ill-formed. [Example:

```
struct Node {
                                    // OK: Refers to Node at global scope
   struct Node* Next:
                                     // OK: Declares type Data
   struct Data* Data;
                                    // at global scope and member Data
 };
 struct Data {
                                    // OK: Refers to Node at global scope
   struct Node* Node;
   friend struct ::Glob;
                                    // error: Glob is not declared
                                    // cannot introduce a qualified type (7.1.6.3)
   friend struct Glob;
                                    // OK: Refers to (as yet) undeclared Glob
                // at global scope.
 struct Base {
                                    // OK: Declares nested Data
   struct Data:
                        thatData; // OK: Refers to ::Data
   struct ::Data*
   struct Base::Data* thisData; // OK: Refers to nested Data
                                    // OK: global Data is a friend
   friend class ::Data;
                                    // OK: nested Data is a friend
   friend class Data;
                                    // Defines nested Data
   struct Data { /* ... */ };
 };
                                    // OK: Redeclares Data at global scope
 struct Data;
                                    // error: cannot introduce a qualified type (7.1.6.3)
 struct :: Data;
 struct Base::Data;
                                    // error: cannot introduce a qualified type (7.1.6.3)
                                    // error: Datum undefined
 struct Base::Datum;
 struct Base::Data* pBase;
                                    // OK: refers to nested Data
— end example]
```

## 3.4.5 Class member access

## [basic.lookup.classref]

- 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 *id-expression* in a class member access (5.2.5) is an *unqualified-id*, and the type of the object expression is of a class type C, the *unqualified-id* is looked up in the scope of class C. For a pseudo-destructor call (5.2.4), the *unqualified-id* is looked up in the context of the complete *postfix-expression*.
- 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:

```
struct A { };

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

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

```
— end example]
```

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

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

the class-name-or-name space-name following the . or  $\rightarrow$  operator is first looked up in the class of the object expression and the name, if found, is used. Otherwise it is looked up in the context of the entire postfix-expression. [Note: See 3.4.3, which describes the lookup of a name before ::, which will only find a type or name space name. — end note]

<sup>5</sup> If the *qualified-id* has the form

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

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

- <sup>6</sup> 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.
- If the *id-expression* is a *conversion-function-id*, its *conversion-type-id* is first looked up in the class of the object expression and the name, if found, is used. Otherwise it is looked up in the context of the entire *postfix-expression*. In each of these lookups, only names that denote types or templates whose specializations are types are considered. [Example:

```
struct A { };
namespace N {
   struct A {
     void g() { }
     template <class T> operator T();
   };
}
int main() {
   N::A a;
   a.operator A();  // calls N::A::operator N::A
}
— end example]
```

#### 3.4.6 Using-directives and namespace aliases

[basic.lookup.udir]

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

## 3.5 Program and linkage

[basic.link]

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

§ 3.5

- a variable, function or function template that is explicitly declared static; or,
- a variable that is explicitly declared **const** or **constexpr** and neither explicitly declared **extern** nor previously declared to have external linkage; or
- a data member of an anonymous union.
- <sup>4</sup> An unnamed namespace or a namespace declared directly or indirectly within an unnamed namespace has internal linkage. All other namespaces have external linkage. A name having namespace scope that has not been given internal linkage above has the same linkage as the enclosing namespace if it is the name of
  - a variable; or
  - a function; 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 linkage; or
  - a template.
- In addition, a member function, static data member, a named class or enumeration of class scope, or an unnamed class or enumeration defined in a class-scope typedef declaration such that the class or enumeration has the typedef name for linkage purposes (7.1.3), has external linkage if the name of the class has external linkage.
- 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:

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

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

§ 3.5

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  $(14)^{33}$ ; 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 cy-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 odr-used (3.2) or is defined in the same translation unit.

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

[Example:

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

§ 3.5 57

<sup>33)</sup> 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.

— end example]

- <sup>9</sup> 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.
- <sup>11</sup> [Note: Linkage to non-C++ declarations can be achieved using a linkage-specification (7.5). end note]

#### 3.6 Start and termination

[basic.start]

## 3.6.1 Main function

[basic.start.main]

- 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 predefine 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:

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

In the latter form argc shall be the number of arguments passed to the program from the environment in which the program is run. If argc is nonzero these arguments shall be supplied in argv[0] through argv[argc-1] as pointers to the initial characters of null-terminated multibyte strings (NTMBS s) (17.5.2.1.4.2) 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]

§ 3.6.1 58

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

- <sup>4</sup> Terminating the program without leaving the current block (e.g., by calling the function std::exit(int) (18.5)) does not destroy any objects with automatic storage duration (12.4). If std::exit is called to end a program during the destruction of an object with static or thread storage duration, the program has undefined behavior.
- A return statement in main has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling std::exit with the return value as the argument. If control reaches the end of main without encountering a return statement, the effect is that of executing

return 0;

#### 3.6.2 Initialization of non-local variables

[basic.start.init]

- 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.
- <sup>2</sup> 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 Ivalue 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 by a constructor call, if the constructor is a constexpr constructor, if all constructor arguments are constant expressions (including conversions), and if, after function invocation substitution (7.1.5), every constructor call and full-expression in the mem-initializers and in the brace-or-equal-initializers for non-static data members is a constant expression;
- if an object with static or thread storage duration is not initialized by a constructor call and if every full-expression that appears in its initializer is a constant expression.

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 unsequenced with respect to the initialization of a variable defined in a different translation unit. Otherwise, the initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of a variable is indeterminately sequenced with respect to every other dynamic initialization. [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

§ 3.6.2 59

<sup>3</sup> 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,

It is implementation-defined whether the dynamic initialization of a non-local variable with static storage duration is done before the first statement of main. If the initialization is deferred to some point in time after the first statement of main, it shall occur before the first odr-use (3.2) of any function or variable defined in the same translation unit as the variable to be initialized.<sup>34</sup> [Example:

```
// - File 1 -
#include "a.h"
#include "b.h"
B b;
A::A(){
  b.Use();
// - File 2 -
#include "a.h"
A a;
// - File 3 -
#include "a.h"
#include "b.h"
extern A a;
extern B b;
int main() {
  a.Use();
  b.Use();
}
```

§ 3.6.2

<sup>34)</sup> A non-local variable with static storage duration having initialization with side-effects must be initialized even if it is not odr-used (3.2, 3.7.1).

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

- It is implementation-defined whether the dynamic initialization of a non-local variable with static or thread storage duration is done before the first statement of the initial function of the thread. If the initialization is deferred to some point in time after the first statement of the initial function of the thread, it shall occur before the first odr-use (3.2) of any variable with thread storage duration defined in the same translation unit as the variable to be initialized.
- <sup>6</sup> If the initialization of a non-local variable with static or thread storage duration exits via an exception, std::terminate is called (15.5.1).

# 3.6.3 Termination [basic.start.term]

- 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. If the destruction of an object with static or thread storage duration exits via an exception, std::terminate is called (15.5.1).
- <sup>2</sup> 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>, 18.5), the call to the function passed to std::atexit is sequenced before the call to the destructor for the object. If a call to std::atexit 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.
- <sup>4</sup> 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]

§ 3.6.3

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

[basic.stc]

- 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
- <sup>2</sup> 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).
- 3 The storage duration categories apply to references as well. The lifetime of a reference is its storage duration.

## 3.7.1 Static storage duration

[basic.stc.static]

- <sup>1</sup> 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).
- <sup>2</sup> 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]
- <sup>4</sup> 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

[basic.stc.thread]

- All variables declared with the thread\_local keyword have thread storage duration. The storage for these entities shall last for the duration of the thread in which they are created. There is a distinct object or reference per thread, and use of the declared name refers to the entity associated with the current thread.
- A variable with thread storage duration shall be initialized before its first odr-use (3.2) and, if constructed, shall be destroyed on thread exit.

#### 3.7.3 Automatic storage duration

[basic.stc.auto]

- Block-scope variables explicitly declared register or not explicitly declared static or extern have automatic storage duration. The storage for these entities lasts until the block in which they are created exits.
- <sup>2</sup> [Note: These variables are initialized and destroyed as described in 6.7. end note]
- <sup>3</sup> 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

[basic.stc.dynamic]

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

§ 3.7.4

The library provides default definitions for the global allocation and deallocation functions. Some global allocation and deallocation functions are replaceable (18.6.1). A C++ program shall provide at most one definition of a replaceable allocation or deallocation function. Any such function definition replaces the default version provided in the library (17.6.4.6). The following allocation and deallocation functions (18.6) are implicitly declared in global scope in each translation unit of a program.

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

These implicit declarations introduce only the function names operator new, operator new[], operator delete, and operator delete[]. [Note: The implicit declarations do not introduce the names std, std::size\_t, or any other names that the library uses to declare these names. Thus, a new-expression, delete-expression or function call that refers to one of these functions without including the header <new> is well-formed. However, referring to std or std::size\_t is ill-formed unless the name has been declared by including the appropriate header. — end note] Allocation and/or deallocation functions can also be declared and defined for any class (12.5).

Any allocation and/or deallocation functions defined in a C++ program, including the default versions in the library, shall conform to the semantics specified in 3.7.4.1 and 3.7.4.2.

#### 3.7.4.1 Allocation functions

#### [basic.stc.dynamic.allocation]

- An allocation function shall be a class member function or a global function; a program is ill-formed if an allocation function is declared in a namespace scope other than global scope or declared static in global scope. The return type shall be void\*. The first parameter shall have type std::size\_t (18.2). The first parameter shall not have an associated default argument (8.3.6). The value of the first parameter shall be interpreted as the requested size of the allocation. An allocation function can be a function template. Such a template shall declare its return type and first parameter as specified above (that is, template parameter types shall not be used in the return type and first parameter type). Template allocation functions shall have two or more parameters.
- The allocation function attempts to allocate the requested amount of storage. If it is successful, it shall return the address of the start of a block of storage whose length in bytes shall be at least as large as the requested size. There are no constraints on the contents of the allocated storage on return from the allocation function. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are unspecified. The pointer returned shall be suitably aligned so that it can be converted to a pointer of any complete object type with a fundamental alignment requirement (3.11) and then used to access the object or array in the storage allocated (until the storage is explicitly deallocated by a call to a corresponding deallocation function). Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned shall be a non-null pointer value (4.10) p0 different from any previously returned value p1, unless that value p1 was subsequently passed to an operator delete. The effect of dereferencing a pointer returned as a request for zero size is undefined.<sup>35</sup>
- An allocation function that fails to allocate storage can invoke the currently installed new-handler function (18.6.2.3), if any. [Note: A program-supplied allocation function can obtain the address of the currently installed new\_handler using the std::get\_new\_handler function (18.6.2.4). —end note] If an allocation function declared with a non-throwing exception-specification (15.4) fails to allocate storage, it shall return a null pointer. Any other allocation function that fails to allocate storage shall indicate failure only by throwing an exception 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

§ 3.7.4.1 63

<sup>35)</sup> The intent is to have operator new() implementable by calling std::malloc() or std::calloc(), so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.

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]

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

- Each deallocation function shall return void and its first parameter shall be void\*. A deallocation function can have more than one parameter. If a class T has a member deallocation function named operator delete with exactly one parameter, then that function is a usual (non-placement) deallocation function. If class T does not declare such an operator delete but does declare a member deallocation function named operator delete with exactly two parameters, the second of which has type std::size\_t (18.2), 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.
- If a deallocation function terminates by throwing an exception, the behavior is undefined. The value of the first argument supplied to a deallocation function may be a null pointer value; if so, and if the deallocation function is one supplied in the standard library, the call has no effect. Otherwise, the behavior is undefined if the value supplied to operator delete(void\*) in the standard library is not one of the values returned by a previous invocation of either operator new(std::size\_t) or operator new(std::size\_t, const std::nothrow\_t&) in the standard library, and the behavior is undefined if the value supplied to operator delete[](void\*) in the standard library is not one of the values returned by a previous invocation of either operator new[](std::size\_t) or operator new[](std::size\_t, const std::nothrow\_t&) in the standard library.
- If the argument given to a deallocation function in the standard library is a pointer that is not the null pointer value (4.10), the deallocation function shall deallocate the storage referenced by the pointer, rendering invalid all pointers referring to any part of the *deallocated storage*. The effect of using an invalid pointer value (including passing it to a deallocation function) is undefined.<sup>36</sup>

#### 3.7.4.3 Safely-derived pointers

[basic.stc.dynamic.safety]

- A traceable pointer object is
  - an object of an object pointer type (3.9.2), or
  - an object of an integral type that is at least as large as std::intptr\_t, or
  - a sequence of elements in an array of character type, where the size and alignment of the sequence match those of some object pointer type.
- <sup>2</sup> A pointer value is a *safely-derived pointer* to a dynamic object only if it has an object pointer type and it is one of the following:
  - the value returned by a call to the C++ standard library implementation of ::operator new(std:: size\_t);<sup>37</sup>

§ 3.7.4.3

<sup>36)</sup> On some implementations, it causes a system-generated runtime fault.

<sup>37)</sup> This section does not impose restrictions on dereferencing pointers to memory not allocated by ::operator new. This maintains the ability of many C++ implementations to use binary libraries and components written in other languages. In particular, this applies to C binaries, because dereferencing pointers to memory allocated by malloc is not restricted.

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

- the result of well-defined pointer arithmetic (5.7) using a safely-derived pointer value;
- the result of a well-defined pointer conversion (4.10, 5.4) of a safely-derived pointer value;
- the result of a reinterpret\_cast of a safely-derived pointer value;
- the result of a 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.
- <sup>3</sup> An integer value is an *integer representation of a safely-derived pointer* only if its type is at least as large as std::intptr\_t and it is one of the following:
  - the result of a 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 reinterpret\_cast<void\*> would compare equal to a safely-derived pointer computable from reinterpret\_cast<void\*>(P).
- An implementation may have relaxed pointer safety, in which case the validity of a pointer value does not depend on whether it is a safely-derived pointer value. Alternatively, an implementation may have strict pointer safety, in which case a pointer value that is not a safely-derived pointer value is an invalid pointer value unless the referenced complete object is of dynamic storage duration and has previously been declared reachable (20.6.4). [Note: the effect of using an invalid pointer value (including passing it to a deallocation function) is undefined, see 3.7.4.2. This is true even if the unsafely-derived pointer value might compare equal to some safely-derived pointer value. end note] It is implementation defined whether an implementation has relaxed or strict pointer safety.

#### 3.7.5 Duration of subobjects

[basic.stc.inherit]

<sup>1</sup> The storage duration of member subobjects, base class subobjects and array elements is that of their complete object (1.8).

### 3.8 Object lifetime

basic.life

- The *lifetime* of an object is a runtime property of the object. 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. [*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
  - if the object has non-trivial initialization, its initialization is complete.

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.

<sup>2</sup> [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
- 4 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<sup>38</sup> 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:

38) For example, before the construction of a global object of non-POD class type (12.7).

- 6 Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. For an object under construction or destruction, see 12.7. Otherwise, such a glvalue refers to allocated storage (3.7.4.2), and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:
  - 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.
- <sup>7</sup> 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;
  void f();
  const C& operator=( const C& );
};

const C& C::operator=( const C& other) {
  if ( this != &other ) {
```

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,<sup>39</sup> the program must ensure that an object of the original type occupies that same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception. [Example:

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

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

- end example]

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

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

const B b;

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

— end example]

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. — $end\ note$ ]

§ 3.8 68

<sup>39)</sup> That is, an object for which a destructor will be called implicitly—upon exit from the block for an object with automatic storage duration, upon exit from the thread for an object with thread storage duration, or upon exit from the program for an object with static storage duration.

3.9 Types [basic.types]

[Note: 3.9 and the subclauses thereof impose requirements on implementations regarding the representation of types. There are two kinds of types: fundamental types and compound types. Types describe objects (1.8), references (8.3.2), or functions (8.3.5). — end note

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

```
#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
                                     // at this point, each subobject of obj of scalar type
 std::memcpy(&obj, buf, N);
                                      // holds its original value
— end example]
```

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

```
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 *tlp contains
    // the same value as the corresponding subobject in *t2p
```

- end example]
- The object representation of an object of type T is the sequence of N unsigned char objects taken up by the object of type T, where N equals size of (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
- 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. <sup>43</sup> Incompletely-defined object types and the void types are incomplete types (3.9.1). Objects shall not be defined to have an incomplete type.
- A class type (such as "class X") might be incomplete at one point in a translation unit and complete later on; the type "class X" is the same type at both points. The declared type of an array object might be an array of incomplete class type and therefore incomplete; if the class type is completed later on in the translation unit, the array type becomes complete; the array type at those two points is the same type. The declared type of an array object might be an array of unknown size and therefore be incomplete at one point in a translation unit and complete later on; the array types at those two points ("array of unknown bound of T" and "array of N T") are different types. The type of a pointer to array of unknown size, or of a type defined by a typedef declaration to be an array of unknown size, cannot be completed. [Example:

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

43) The size and layout of an instance of an incompletely-defined object type is unknown.

<sup>40)</sup> By using, for example, the library functions (17.6.1.2) std::memcpy or std::memmove.

<sup>41)</sup> By using, for example, the library functions (17.6.1.2) std::memcpy or std::memmove.

<sup>42)</sup> The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.

```
// UNKA is an incomplete type
typedef int UNKA[];
UNKA* arrp;
                                    // arrp is a pointer to an incomplete type
UNKA** arrpp;
void foo() {
                                    // ill-formed: X is incomplete
  xp++;
  arrp++;
                                    // ill-formed: incomplete type
                                    // OK: sizeof UNKA* is known
  arrpp++;
struct X { int i; };
                                    // now X is a complete type
                                    // now the type of arr is complete
int arr[10];
X x;
void bar() {
                                    // OK; type is "pointer to X"
  xp = &x;
                                    // ill-formed: different types
  arrp = &arr;
                                    // OK: X is complete
  xp++;
                                    // ill-formed: UNKA can't be completed
  arrp++;
```

- end example]
- <sup>7</sup> [Note: The rules for declarations and expressions describe in which contexts incomplete types are prohibited.  $-end\ note$ ]
- <sup>8</sup> An *object type* is a (possibly cv-qualified) type that is not a function type, not a reference type, and not a void type.
- Arithmetic types (3.9.1), enumeration types, pointer types, pointer to member types (3.9.2), 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 reference type referring to a literal type; or
  - an array of literal type; or
  - a class type (Clause 9) that has all of the following properties:
    - it has a trivial destructor.
    - every constructor call and full-expression in the *brace-or-equal-initializers* for non-static data members (if any) is a constant expression (5.19),
    - it is an aggregate type (8.5.1) or has at least one constructor constructor template that is not a copy or move constructor, and
    - all of its non-static data members and base classes are of literal types.
- <sup>11</sup> 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

## [basic.fundamental]

- Objects declared as characters (char) shall be large enough to store any member of the implementation's basic character set. If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character. It is implementation-defined whether a char object can hold negative values. Characters can be explicitly declared unsigned or signed. Plain char, signed char, and unsigned char are three distinct types. A char, a signed char, and an unsigned char occupy the same amount of storage and have the same alignment requirements (3.11); that is, they have the same object representation. For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types. In any particular implementation, a plain char object can take on either the same values as a signed char or an unsigned char; which one is implementation-defined.
- There are five standard signed integer types: "signed char", "short int", "int", "long int", and "long long int". In this list, each type provides at least as much storage as those preceding it in the list. There may also be implementation-defined extended signed integer types. The standard and extended signed integer types are collectively called signed integer types. Plain ints have the natural size suggested by the architecture of the execution environment<sup>44</sup>; the other signed integer types are provided to meet special needs.
- For each of the standard signed integer types, there exists a corresponding (but different) standard unsigned integer type: "unsigned char", "unsigned short int", "unsigned int", "unsigned long int", and "unsigned long long int", each of which occupies the same amount of storage and has the same alignment requirements (3.11) as the corresponding signed integer type <sup>45</sup>; 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.
- <sup>4</sup> 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. <sup>46</sup>
- Type wchar\_t is a distinct type whose values can represent distinct codes for all members of the largest extended character set specified among the supported locales (22.3.1). Type wchar\_t shall have the same size, signedness, and alignment requirements (3.11) as one of the other integral types, called its *underlying type*. Types char16\_t and char32\_t denote distinct types with the same size, signedness, and alignment as uint\_least16\_t and uint\_least32\_t, respectively, in <stdint.h>, called the underlying types.
- Values of type bool are either true or false. 47 [Note: There are no signed, unsigned, short, or long bool types or values. end note] Values of type bool participate in integral promotions (4.5).
- <sup>7</sup> Types bool, char, char16\_t, char32\_t, wchar\_t, and the signed and unsigned integer types are collectively called *integral* types. <sup>48</sup> A synonym for integral type is *integer type*. The representations of integral types shall define values by use of a pure binary numeration system. <sup>49</sup> [Example: this International Standard

§ 3.9.1 71

\_\_\_\_

<sup>44)</sup> that is, large enough to contain any value in the range of INT\_MIN and INT\_MAX, as defined in the header <climits>.

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

<sup>46)</sup> 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.

<sup>47)</sup> Using a 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 true nor false.

<sup>48)</sup> Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to integral types as specified in 4.5.

<sup>49)</sup> A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive

- permits 2's complement, 1's complement and signed magnitude representations for integral types. -end example
- There are three *floating point* types: float, double, and long double. The type double provides at least as much precision as float, and the type long double provides at least as much precision as double. The set of values of the type float is a subset of the set of values of the type double; the set of values of the type double is a subset of the set of values of the type long double. The value representation of floating-point types is implementation-defined. *Integral* and *floating* types are collectively called *arithmetic* types. Specializations of the standard template std::numeric\_limits (18.3) shall specify the maximum and minimum values of each arithmetic type for an implementation.
- The void type has an empty set of values. The void type is an incomplete type that cannot be completed. It is used as the return type for functions that do not return a value. Any expression can be explicitly converted to type cv void (5.4). An expression of type void shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18), as a second or third operand of ?: (5.16), as the operand of typeid or decltype, as the expression in a return statement (6.6.3) for a function with the return type void, or as the operand of an explicit conversion to type cv void.
- A value of type std::nullptr\_t is a null pointer constant (4.10). Such values participate in the pointer and the pointer to member conversions (4.10, 4.11). sizeof(std::nullptr\_t) shall be equal to sizeof(void\*).
- <sup>11</sup> [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

[basic.compound]

- Compound types can be constructed in the following ways:
  - arrays of objects of a given type, 8.3.4;
  - functions, which have parameters of given types and return void or references or objects of a given type, 8.3.5;
  - pointers to void or objects or functions (including static members of classes) of a given type, 8.3.1;
  - references to objects or functions of a given type, 8.3.2. There are two types of references:
    - lvalue reference
    - rvalue reference
  - classes containing a sequence of objects of various types (Clause 9), a set of types, enumerations and functions for manipulating these objects (9.3), and a set of restrictions on the access to these entities (Clause 11);
  - unions, which are classes capable of containing objects of different types at different times, 9.5;
  - *enumerations*, which comprise a set of named constant values. Each distinct enumeration constitutes a different *enumerated type*, 7.2;
  - pointers to non-static <sup>50</sup> class members, which identify members of a given type within objects of a given class, 8.3.3.
- <sup>2</sup> 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.
- <sup>3</sup> The type of a pointer to void or a pointer to an object type is called an *object pointer type*. [Note: A pointer to void does not have a pointer-to-object type, however, because void is not an object type. end note] The type of a pointer that can designate a function is called a function pointer type. A pointer to objects

bits are additive, begin with 1, and are multiplied by successive integral power of 2, except perhaps for the bit with the highest position. (Adapted from the American National Dictionary for Information Processing Systems.)

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

§ 3.9.2

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 cy-qualified and cy-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]

<sup>4</sup> A pointer to *cv*-qualified (3.9.3) or *cv*-unqualified void can be used to point to objects of unknown type. Such a pointer shall be able to hold any object pointer. An object of type *cv* void\* shall have the same representation and alignment requirements as *cv* char\*.

# 3.9.3 CV-qualifiers

# [basic.type.qualifier]

- 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).<sup>51</sup>
- 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*.
- <sup>4</sup> There is a partial ordering on cv-qualifiers, so that a type can be said to be *more cv-qualified* than another. Table 9 shows the relations that constitute this ordering.
- In this International Standard, the notation cv (or cv1, cv2, etc.), used in the description of types, represents an arbitrary set of cv-qualifiers, i.e., one of {const}, {volatile}, {const, volatile}, or the empty set. Cv-qualifiers applied to an array type attach to the underlying element type, so the notation "cv T," where T is an array type, refers to an array whose elements are so-qualified. Such array types can be said to be

§ 3.9.3 73

Table 9 — Relations on const and volatile

1.0		
no cv-qualifier	<	const
no cv-qualifier	<	volatile
no cv-qualifier	<	const volatile
const	<	const volatile
volatile	<	const volatile

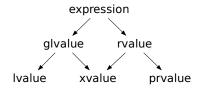


Figure 1 — Expression category taxonomy

more (or less) cy-qualified than other types based on the cy-qualification of the underlying element types.

#### 3.10 Lvalues and rvalues

[basic.lval]

- Expressions are categorized according to the taxonomy in Figure 1.
  - 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 expression) is an xvalue, a temporary object (12.2) or subobject thereof, or a value that is not associated with an object.
  - A prvalue ("pure" rvalue) is an rvalue that is not an xvalue. [Example: The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as 12, 7.3e5, or true is also a prvalue. end example]

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

§ 3.10 74

<sup>51)</sup> 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.

functions, and the categories of values they expect and yield are determined by their parameter and return types.  $-end\ note$ 

- Whenever a glvalue appears in a context where a prvalue is expected, the glvalue is converted to a prvalue; see 4.1, 4.2, and 4.3. [Note: An attempt to bind an rvalue reference to an lvalue is not such a context; see 8.5.3. end note]
- <sup>3</sup> 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.
- <sup>4</sup> Class prvalues can have cv-qualified types; non-class prvalues always have cv-unqualified types. Unless otherwise indicated (5.2.2), prvalues shall always have complete types or the void type; in addition to these types, glvalues can also have incomplete types.
- An Ivalue 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]
- <sup>6</sup> Functions cannot be modified, but pointers to functions can be modifiable.
- <sup>7</sup> 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.
- <sup>8</sup> 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).
- <sup>9</sup> 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.
- <sup>10</sup> If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined:<sup>52</sup>
  - the dynamic type of the object,
  - a cv-qualified version of the dynamic type of the object,
  - a type similar (as defined in 4.4) to the dynamic type of the object,
  - a type that is the signed or unsigned type corresponding to the dynamic type of the object,
  - a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,
  - an aggregate or union type that includes one of the aforementioned types among its elements or non-static data members (including, recursively, an element or non-static data member of a subaggregate or contained union),
  - a type that is a (possibly cy-qualified) base class type of the dynamic type of the object,
  - a char or unsigned char type.

# 3.11 Alignment [basic.align]

- Object types have alignment requirements (3.9.1, 3.9.2) which place restrictions on the addresses at which an object of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using the alignment specifier (7.6.2).
- A fundamental alignment is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to alignof(std::max\_align\_t) (18.2). The alignment required for a type might be different when it is used as the type of a complete object and when it is used as the type of a subobject. [Example:

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

§ 3.11 75

```
struct B { long double d; };
struct D : virtual B { char c; }
```

When D is the type of a complete object, it will have a subobject of type B, so it must be aligned appropriately for a long double. If D appears as a subobject of another object that also has B as a virtual base class, the B subobject might be part of a different subobject, reducing the alignment requirements on the D subobject. — end example] The result of the alignof operator reflects the alignment requirement of the type in the complete-object case.

- <sup>3</sup> An extended alignment is represented by an alignment greater than alignof(std::max\_align\_t). It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported (7.6.2). A type having an extended alignment requirement is an over-aligned type. [Note: every over-aligned type is or contains a class type to which extended alignment applies (possibly through a non-static data member). end note]
- 4 Alignments are represented as values of the type std::size\_t. Valid alignments include only those values returned by an alignof expression for the fundamental types plus an additional implementation-defined set of values, which may be empty. Every alignment value shall be a non-negative integral power of two.
- <sup>5</sup> 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]
- 7 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.
- <sup>8</sup> [Note: The runtime pointer alignment function (20.6.5) can be used to obtain an aligned pointer within a buffer; the aligned-storage templates in the library (20.9.7.6) can be used to obtain aligned storage. end note ]
- <sup>9</sup> 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.

§ 3.11 76

# 4 Standard conversions

[conv]

Standard conversions are implicit conversions with built-in meaning. Clause 4 enumerates the full set of such conversions. A *standard conversion sequence* is a sequence of standard conversions in the following order:

- 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.

- <sup>2</sup> [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]
- 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 Ivalue if T is an Ivalue 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.
- <sup>4</sup> [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]
- <sup>5</sup> [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]

Standard conversions 77

#### 4.1 Lvalue-to-rvalue conversion

[conv.lval]

A glvalue (3.10) of a non-function, non-array type T can be converted to a prvalue.<sup>53</sup> 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.<sup>54</sup>

- When an Ivalue-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 std::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

[conv.array]

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

[conv.func]

- 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.<sup>55</sup>
- <sup>2</sup> [Note: See 13.4 for additional rules for the case where the function is overloaded. -end note]

## 4.4 Qualification conversions

[conv.qual]

- A prvalue of type "pointer to cv1 T" can be converted to a prvalue of type "pointer to cv2 T" if "cv2 T" is more cv-qualified than "cv1 T".
- A prvalue of type "pointer to member of X of type cv1 T" can be converted to a prvalue of type "pointer to member of X of type cv2 T" if "cv2 T" is more cv-qualified than "cv1 T".
- 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:  $^{56}$

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

T1 is  $cv_{1,0}$  pointer to  $cv_{1,1}$  pointer to  $\cdots$   $cv_{1,n-1}$  pointer to  $cv_{1,n}$  T

and

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

where each  $cv_{i,j}$  is const, volatile, const volatile, or nothing. The n-tuple of cv-qualifiers after the first in a pointer type, e.g.,  $cv_{1,1}$ ,  $cv_{1,2}$ ,  $\cdots$ ,  $cv_{1,n}$  in the pointer type T1, is called the cv-qualification signature of the pointer type. An expression of type T1 can be converted to type T2 if and only if the following conditions are satisfied:

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

§ 4.4 78

<sup>53)</sup> 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.

<sup>54)</sup> 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.

<sup>55)</sup> This conversion never applies to non-static member functions because an Ivalue that refers to a non-static member function cannot be obtained.

<sup>56)</sup> These rules ensure that const-safety is preserved by the conversion.

```
— if the cv_{1,j} and cv_{2,j} are different, then const is in every cv_{2,k} for 0 < k < j.
```

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

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

```
cv_0P_0 to cv_1P_1 to \cdots cv_{n-1}P_{n-1} to cv_n T
```

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

Two multi-level pointer to member types or two multi-level mixed pointer and pointer to member types T1 and T2 are similar if there exists a type T and integer n > 0 such that:

```
T1 is cv_{1,0}P_0 to cv_{1,1}P_1 to \cdots cv_{1,n-1}P_{n-1} to cv_{1,n} T and T2 is cv_{2,0}P_0 to cv_{2,1}P_1 to \cdots cv_{2,n-1}P_{n-1} to cv_{2,n} T
```

<sup>7</sup> 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

-end note

[conv.prom]

- A prvalue of an integer type other than bool, char16\_t, char32\_t, or wchar\_t whose integer conversion rank (4.13) is less than the rank of int can be converted to a prvalue of type int if int can represent all the values of the source type; otherwise, the source prvalue can be converted to a prvalue of type unsigned int.
- A prvalue of type char16\_t, char32\_t, or wchar\_t (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: int, unsigned int, long int, unsigned long int, long int, or unsigned long long int. If none of the types in that list can represent all the values of its underlying type, a prvalue of type char16\_t, char32\_t, or wchar\_t can be converted to a prvalue of its underlying type.
- A prvalue of an unscoped enumeration type whose underlying type is not fixed (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  $b_{min}$  to  $b_{max}$  as described in 7.2): int, unsigned int, long int, unsigned long int, long long int, or unsigned long long int. If none of the types in that list can represent all the values of the enumeration, 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.
- <sup>4</sup> 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

§ 4.5

prvalue of an unscoped enumeration type whose underlying type is fixed can also be converted to a prvalue of the promoted underlying type.

- 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.
- <sup>6</sup> A prvalue of type bool can be converted to a prvalue of type int, with false becoming zero and true becoming one.
- <sup>7</sup> These conversions are called *integral promotions*.

# 4.6 Floating point promotion

[conv.fpprom]

- 1 A prvalue of type float can be converted to a prvalue of type double. The value is unchanged.
- <sup>2</sup> This conversion is called *floating point promotion*.

# 4.7 Integral conversions

[conv.integral]

- 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.
- <sup>2</sup> 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]
- <sup>3</sup> 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.
- <sup>4</sup> 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.
- <sup>5</sup> The conversions allowed as integral promotions are excluded from the set of integral conversions.

## 4.8 Floating point conversions

[conv.double]

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

### 4.9 Floating-integral conversions

[conv.fpint]

- A prvalue of a floating point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type. [Note: If the destination type is bool, see 4.12. end note]
- A prvalue of an integer type or of an unscoped enumeration type can be converted to a prvalue of a floating point type. The result is exact if possible. If the value being converted is in the range of values that can be represented but the value cannot be represented exactly, it is an implementation-defined choice of either the next lower or higher representable value. [Note: Loss of precision occurs if the integral value cannot be represented exactly as a value of the floating type. —end note] If the value being converted is outside the range of values that can be represented, the behavior is undefined. If the source type is bool, the value false is converted to zero and the value true is converted to one.

#### 4.10 Pointer conversions

[conv.ptr]

A null pointer constant is an integral constant expression (5.19) prvalue of integer type that evaluates to zero or a prvalue of type std::nullptr\_t. A null pointer constant can be converted to a pointer type; the result is the null pointer value of that type and is distinguishable from every other value of object pointer or function pointer type. Such a conversion is called a null pointer conversion. Two null pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to cv-qualified type is a single conversion, and not the sequence of a pointer conversion followed by a qualification conversion (4.4).

§ 4.10

A null pointer constant of integral type can be converted to a prvalue of type  $std::nullptr_t$ . [Note: The resulting prvalue is not a null pointer value. — end note]

- 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.
- <sup>3</sup> 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

[conv.mem]

- A null pointer constant (4.10) can be converted to a pointer to member type; the result is the *null member* pointer value of that type and is distinguishable from any pointer to member not created from a null pointer constant. Such a conversion is called a *null member pointer conversion*. Two null member pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to member of cv-qualified type is a single conversion, and not the sequence of a pointer to member conversion followed by a qualification conversion (4.4).
- 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. 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. <sup>57</sup>

#### 4.12 Boolean conversions

[conv.bool]

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

[conv.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 signed char.
  - The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type.

§ 4.13

<sup>57)</sup> 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 an object pointer or a function pointer and the rules for conversions of such pointers do not apply to pointers to members. In particular, a pointer to member cannot be converted to a void\*.

— 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 (Clause 5). -end note]

§ 4.13 82

# 5 Expressions

[expr]

<sup>1</sup> [Note: Clause 5 defines the syntax, order of evaluation, and meaning of expressions.<sup>58</sup> An expression is a sequence of operators and operands that specifies a computation. An expression can result in a value and can cause side effects. — end note]

- [Note: Operators can be overloaded, that is, given meaning when applied to expressions of class type (Clause 9) or enumeration type (7.2). Uses of overloaded operators are transformed into function calls as described in 13.5. Overloaded operators obey the rules for syntax specified in Clause 5, but the requirements of operand type, 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]
- <sup>3</sup> 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.
- <sup>4</sup> 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]
- 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.
- <sup>6</sup> [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 of non-reference type 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:
struct A {
   int m;
};
A&& operator+(A, A);
A&& f();
```

Expressions 83

<sup>58)</sup> The precedence of operators is not directly specified, but it can be derived from the syntax.

```
A a;
A&& ar = static_cast<A&&>(a);
```

The expressions f(), f().m, static\_cast<A&&>(a), and a + a are xvalues. The expression ar is an lvalue.  $-end\ example$ ]

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 const int can, for example, be used where a prvalue expression of type 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:
  - 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.
  - If either operand is of type long double, the other shall be converted to long double.
  - Otherwise, if either operand is double, the other shall be converted to double.
  - Otherwise, if either operand is float, the other shall be converted to float.
  - Otherwise, the integral promotions (4.5) shall be performed on both operands.<sup>59</sup> Then the following rules shall be applied to the promoted operands:
    - If both operands have the same type, no further conversion is needed.
    - Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank shall be converted to the type of the operand with greater rank.
    - Otherwise, if the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type shall be converted to the type of the operand with unsigned integer type.
    - Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type shall be converted to the type of the operand with signed integer type.
    - Otherwise, both operands shall be converted to the unsigned integer type corresponding to the type of the operand with signed integer type.
- In some contexts, an expression only appears for its side effects. Such an expression is called a *discarded-value expression*. The expression is evaluated and its value is discarded. The array-to-pointer (4.2) and function-to-pointer (4.3) standard conversions are not applied. The lvalue-to-rvalue conversion (4.1) is applied only if the expression is an lvalue of volatile-qualified type and it has one of the following forms:

```
-id-expression (5.1.1),
```

— subscripting (5.2.1),

Expressions 84

<sup>59)</sup> As a consequence, operands of type bool, char16\_t, char32\_t, wchar\_t, or an enumerated type are converted to some integral type.

```
class member access (5.2.5),
indirection (5.3.1),
pointer-to-member operation (5.5),
conditional expression (5.16) where both the second and the third operands are one of the above, or
comma expression (5.18) where the right operand is one of the above.
```

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.<sup>60</sup>

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

```
struct A {
  char g();
  template<class T> auto f(T t) -> decltype(t + g())
      { return t + g(); }
};
template auto A::f(int t) -> decltype(t + g());
```

<sup>60)</sup> The cast and assignment operators must still perform their specific conversions as described in 5.4, 5.2.9 and 5.17.

- end example]
- 4 Otherwise, if a *member-declarator* declares a non-static data member (9.2) of a class X, the expression this is a prvalue of type "pointer to X" within the optional *brace-or-equal-initializer*. It shall not appear elsewhere in the *member-declarator*.
- The expression this shall not appear in any other context. [Example:

— end example]

- A parenthesized expression is a primary expression whose type and value are identical to those of the enclosed expression. The presence of parentheses does not affect whether the expression is an Ivalue. The parenthesized expression can be used in exactly the same contexts as those where the enclosed expression can be used, and with the same meaning, except as otherwise indicated.
- An id-expression is a restricted form of a primary-expression. [Note: an id-expression can appear after and  $\rightarrow$  operators (5.2.5). end note]
- An identifier is an id-expression provided it has been suitably declared (Clause 7). [Note: for operator-function-ids, see 13.5; for conversion-function-ids, see 12.3.2; for literal-operator-ids, see 13.5.8; for template-ids, see 14.2. A class-name or decltype-specifier prefixed by ~ denotes a destructor; see 12.4. Within the definition of a non-static member function, an identifier that names a non-static member is transformed to a class member access expression (9.3.1). end note] The type of the expression is the type of the identifier. The result is the entity denoted by the identifier. The result is an Ivalue if the entity is a function, variable, or data member and a prvalue otherwise.

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

nested-name-specifier:
    :: opt type-name ::
    :: opt namespace-name ::
    decltype-specifier ::
    nested-name-specifier identifier ::
    nested-name-specifier templateopt 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 Ivalue 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  $\sim$  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]

- <sup>9</sup> A::, or a nested-name-specifier that names a namespace (7.3), in either case 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.
- 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 unqualified-id is a conversion-function-id, its conversion-type-id shall denote the same type in both the context in which the entire qualified-id occurs and in the context of the class denoted by the nested-name-specifier.
- 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<sup>61</sup> or a class derived from that class, or
  - to form a pointer to member (5.3.1), 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:

```
struct S {
  int m;
```

<sup>61)</sup> This also applies when the object expression is an implicit (\*this) (9.3.1).

## 5.1.2 Lambda expressions

[expr.prim.lambda]

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

```
#include <algorithm>
 #include <cmath>
 void abssort(float *x, unsigned N) {
    std::sort(x, x + N,
      [](float a, float b) {
        return std::abs(a) < std::abs(b);
      });
 }
— end example]
         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 \dots opt
               capture-list, capture ..._{opt}
         capture:
               identifier
               & identifier
               this
         lambda-declarator:
               ( parameter-declaration-clause ) mutable opt
                      exception-specificationopt attribute-specifier-seqopt trailing-return-typeopt
```

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)] and note.]

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:

- 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 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. An attribute-specifier-seq in a lambda-declarator appertains 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]
- 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 idexpressions 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:

— end example]

<sup>8</sup> 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:

— end example]

- <sup>9</sup> 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]
- 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*.
- If a lambda-expression has an associated capture-default and its compound-statement odr-uses (3.2) this or a variable with automatic storage duration and the odr-used entity is not explicitly captured, then the odr-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 odr-uses of this can result in implicit capture. end note]
- An entity is captured if it is captured explicitly or implicitly. An entity captured by a lambda-expression is odr-used (3.2) in the scope containing the lambda-expression. If this is captured by a local lambda expression, its nearest enclosing function shall be a non-static member function. If a lambda-expression odr-uses (3.2) this or a variable with automatic storage duration from its reaching scope, that entity shall be captured by the lambda-expression. If a lambda-expression captures an entity and that entity is not defined or captured in the immediately enclosing lambda expression or function, the program is ill-formed. [Example:

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

— end example]

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

— end example]

- 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 m2 captures an entity and that entity is captured by an immediately enclosing lambda-expression m1, then m2's capture is transformed as follows:
  - if m1 captures the entity by copy, m2 captures the corresponding non-static data member of m1's closure type;
  - if m1 captures the entity by reference, m2 captures the same entity captured by m1.

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

```
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;</pre>
```

```
— end example]
```

17 Every *id-expression* that is an odr-use (3.2) of an entity captured by copy is transformed into an access to the corresponding unnamed data member of the closure type. [Note: An *id-expression* that is not an odr-use refers to the original entity, never to a member of the closure type. Furthermore, such an *id-expression* does not cause the implicit capture of the entity. —end note] If this is captured, each odr-use of this is transformed into an access to the corresponding unnamed data member of the closure type, cast (5.4) to the type of this. [Note: The cast ensures that the transformed expression is a prvalue. —end note] [Example:

— end example]

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

— end example]

- The closure type associated with a *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). [Note: The copy/move constructor is implicitly defined in the same way as any other implicitly declared copy/move constructor would be implicitly defined. —end note]
- <sup>20</sup> The closure type associated with a *lambda-expression* has an implicitly-declared destructor (12.4).
- When the *lambda-expression* is evaluated, the entities that are captured by copy are used to direct-initialize each corresponding non-static data member of the resulting closure object. (For array members, the array elements are direct-initialized in increasing subscript order.) These initializations are performed in the (unspecified) order in which the non-static data members are declared. [*Note:* This ensures that the destructions will occur in the reverse order of the constructions. end note]
- [Note: If an entity is implicitly or explicitly captured by reference, invoking the function call operator of the corresponding lambda-expression after the lifetime of the entity has ended is likely to result in undefined behavior. end note]
- <sup>23</sup> A capture followed by an ellipsis is a pack expansion (14.5.3). [Example:

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

```
lm();
}
— end example]
```

## 5.2 Postfix expressions

[expr.post]

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\mbox{-specifier braced-init-list}
      postfix\text{-}expression . template_{opt} id\text{-}expression
      postfix-expression -> template<sub>opt</sub> id-expression
      post fix\hbox{-}expression\ .\ pseudo\hbox{-}destructor\hbox{-}name
      postfix-expression -> pseudo-destructor-name
      postfix-expression ++
      postfix-expression --
      dynamic_cast < type-id > (expression)
      static_cast < type-id > ( expression )
      reinterpret_cast < type-id > ( expression )
      const_cast < type-id > ( expression )
      typeid ( expression )
      typeid ( type-id )
expression-list:
      initializer-list
pseudo-destructor-name:
      nested-name-specifier_{opt} type-name :: \sim type-name
      nested-name-specifier template simple-template-id :: ~ type-name
      nested-name-specifier_{opt} \sim type-name
       ~ decltype-specifier
```

[Note: The > token following the type-id in a dynamic\_cast, static\_cast, reinterpret\_cast, or const\_cast may be the product of replacing a >> token by two consecutive > tokens (14.2). — end note]

#### 5.2.1 Subscripting

[expr.sub]

- 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 Ivalue of type "T." The type "T" shall be a completely-defined object type. <sup>62</sup> The expression E1[E2] is identical (by definition) to \*((E1)+(E2)) [Note: see 5.3 and 5.7 for details of \* and + and 8.3.4 for details of arrays. end note]
- <sup>2</sup> A braced-init-list shall not be used with the built-in subscript operator.

# 5.2.2 Function call [expr.call]

There are two kinds of function call: ordinary function call and member function <sup>63</sup> (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 Ivalue 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

§ 5.2.2 93

<sup>62)</sup> This is true even if the subscript operator is used in the following common idiom: &x[0].

<sup>63)</sup> A static member function (9.4) is an ordinary function.

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 *idexpression* is a function member name, or a pointer-to-member expression (5.5) selecting a function member; the call is as a member of the class object referred to by the object expression. In the case of an implicit class member access, the implied object is the one pointed to by this. [Note: a member function call of the form f() is interpreted as (\*this).f() (see 9.3.1). — end note] If a function or member function name is used, the name can be overloaded (Clause 13), in which case the appropriate function shall be selected according to the rules in 13.3. 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. [Note: the dynamic type is the type of the object referred to by the current value of the object expression. 12.7 describes the behavior of virtual function calls when the object expression refers to an object under construction or destruction. — end note]

- <sup>2</sup> [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]
- If the postfix-expression designates a destructor (12.4), the type of the function call expression is void; otherwise, the type of the function call expression is the return type of the statically chosen function (i.e., ignoring the virtual keyword), even if the type of the function actually called is different. This type shall be an object type, a reference type or the type void.
- When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument. [Note: Such initializations are indeterminately sequenced with respect to each other (1.9) – end note 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.
- [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]
- When there is no parameter for a given argument, the argument is passed in such a way that the receiving

§ 5.2.2

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) having a non-trivial copy constructor, a non-trivial move constructor, or a non-trivial destructor, with no corresponding parameter, is conditionally-supported with implementation-defined semantics. If the argument has integral or enumeration type that is subject to the integral promotions (4.5), or a floating point type that is subject to the floating point promotion (4.6), the value of the argument promotions.

- 8 [Note: The evaluations of the postfix expression and of the argument expressions are all unsequenced relative to one another. All side effects of argument expression evaluations are sequenced before the function is entered (see 1.9). end note]
- 9 Recursive calls are permitted, except to the function named main (3.6.1).
- A function call is an Ivalue if the result type is an Ivalue 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.
- 11 If a function call is a prvalue of object type:
  - if the function call is either
    - the operand of a decltype-specifier or
    - the right operand of a comma operator that is the operand of a decltype-specifier,

a temporary object is not introduced for the prvalue. The type of the prvalue may be incomplete. [Note: as a result, storage is not allocated for the prvalue and it is not destroyed; thus, a class type is not instantiated as a result of being the type of a function call in this context. This is true regardless of whether the expression uses function call notation or operator notation (13.3.1.2). — end note] [Note: unlike the rule for a decltype-specifier that considers whether an id-expression is parenthesized (7.1.6.2), parentheses have no special meaning in this context. — end note]

— otherwise, the type of the prvalue shall be complete.

## 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 expression (5.4). If the type specified is a class type, the class type shall be complete. If the expression list specifies more than a single value, the type shall be a class with a suitably declared constructor (8.5, 12.1), and the expression T(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 object 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

nested-name- $specifier_{opt}$  type-name ::  $\sim$  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 (14.2), and then followed by an *id-expression*, is a postfix expression. The postfix expression before the dot or arrow is evaluated;<sup>64</sup> the result of that evaluation, together with the *id-expression*, determines the result of the entire postfix expression.
- <sup>2</sup> For the first option (dot) the first expression shall have complete class type. For the second option (arrow) the first expression shall have pointer to complete class type. The expression E1->E2 is converted to the equivalent form (\*(E1)).E2; the remainder of 5.2.5 will address only the first option (dot).<sup>65</sup> In either case, the *id-expression* shall name a member of the class or of one of its base classes. [*Note:* because the name of a class is inserted in its class scope (Clause 9), the name of a class is also considered a nested member of that class. *end note*] [*Note:* 3.4.5 describes how names are looked up after the . and -> operators. *end note*]
- 3 Abbreviating postfix-expression.id-expression as E1.E2, E1 is called the object expression. The type and value category of E1.E2 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.
- <sup>4</sup> 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 Ivalue, then E1.E2 is an Ivalue; 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 "cq12 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 Ivalue; the expression designates the static member function. The

65) Note that (\*(E1)) is an lvalue.

<sup>64)</sup> 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.

- 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.
- <sup>5</sup> 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. [Note: The program is also ill-formed if the naming class is an ambiguous base of the class type of the object expression; see 11.2. end note]

#### 5.2.6 Increment and decrement

## [expr.post.incr]

- 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.
- 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]

- 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).
- <sup>2</sup> 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.
- <sup>3</sup> 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).
- <sup>4</sup> If the value of v is a null pointer value in the pointer case, the result is the null pointer value of type T.
- If T is "pointer to cv1 B" and v has type "pointer to cv2 D" such that B is a base class of D, the result is a pointer to the unique B subobject of the D object pointed to by v. Similarly, if T is "reference to cv1 B" and v has type cv2 D such that B is a base class of D, the result is the unique B subobject of the D object referred to by v. <sup>66</sup> The result is an Ivalue if T is an Ivalue 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:

<sup>66)</sup> The most derived object (1.8) pointed or referred to by v can contain other B objects as base classes, but these are ignored.

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

— end example]

- Otherwise, v shall be a pointer to or an lvalue of a polymorphic type (10.3).
- <sup>7</sup> If T is "pointer to *cv* void," then the result is a pointer to the most derived object pointed to by v. Otherwise, a run-time check is applied to see if the object pointed or referred to by v can be converted to the type pointed or referred to by T.
- <sup>8</sup> If C is the class type to which T points or refers, the run-time check logically executes as follows:
  - If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class subobject of a 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.
- 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:

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
    d;
  D
                                      // cast needed to break protection
 B* bp = (B*)&d;
                                      // public derivation, no cast needed
 A* ap = &d;
 D& dr = dynamic_cast<D&>(*bp);
                                     // fails
                                     // fails
 ap = dynamic_cast<A*>(bp);
  bp = dynamic_cast<B*>(ap);
                                     // fails
                                     // succeeds
  ap = dynamic_cast<A*>(&d);
                                     // ill-formed (not a run-time check)
 bp = dynamic_cast<B*>(&d);
class E : public D, public B { };
class F : public E, public D { };
void h() {
 F f;
                                      // succeeds: finds unique A
 A* ap = &f;
 D* dp = dynamic_cast<D*>(ap);
                                      // fails: yields 0
                                      // f has two D subobjects
 E* ep = (E*)ap;
                                     // ill-formed: cast from virtual base
  E* ep1 = dynamic_cast<E*>(ap);
                                     // succeeds
```

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

## 5.2.8 Type identification

[expr.typeid]

The result of a typeid expression is an Ivalue 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.<sup>67</sup> The lifetime of the object referred to by the Ivalue 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.

- When typeid is applied to a glvalue expression whose type is a polymorphic class type (10.3), the result refers to a std::type\_info object representing the type of the most derived object (1.8) (that is, the dynamic type) to which the glvalue refers. If the glvalue expression is obtained by applying the unary \* operator to a pointer<sup>68</sup> and the pointer is a null pointer value (4.10), the typeid expression throws the std::bad\_typeid exception (18.7.3).
- 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).
- 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.
- The top-level cv-qualifiers of the glvalue expression or the *type-id* that is the operand of **typeid** are always ignored. [Example:

- <sup>6</sup> If the header <typeinfo> (18.7.1) is not included prior to a use of typeid, the program is ill-formed.
- <sup>7</sup> [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]

- The result of the expression static\_cast<T>(v) is the result of converting the expression v to type T. If T is an Ivalue reference type or an rvalue reference to function type, the result is an Ivalue; 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).
- An Ivalue 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 Ivalue 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:

```
struct B { };
```

<sup>67)</sup> The recommended name for such a class is extended\_type\_info.

<sup>68)</sup> If p is an expression of pointer type, then \*p, (\*p), \*(p), ((\*p)), \*((p)), and so on all meet this requirement.

```
struct D : public B { };
D d;
B &br = d;
static_cast<D&>(br);  // produces lvalue to the original d object
— end example]
```

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.
- <sup>6</sup> Any expression can be explicitly converted to type cv void, in which case it becomes a discarded-value expression (Clause 5). [Note: however, if the value is in a temporary object (12.2), the destructor for that object is not executed until the usual time, and the value of the object is preserved for the purpose of executing the destructor. end note]
- <sup>7</sup> The inverse of any standard conversion sequence (Clause 4) not containing an Ivalue-to-rvalue (4.1), array-to-pointer (4.2), function-to-pointer (4.3), null pointer (4.10), null member pointer (4.11), or boolean (4.12) conversion, can be performed explicitly using static\_cast. A program is ill-formed if it uses static\_cast to perform the inverse of an ill-formed standard conversion sequence. [Example:

- end example]
- <sup>8</sup> 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 value is unspecified (and might not be in that range). A value of floating-point type can also be converted to an enumeration type. The resulting value is the same as converting the original value to the underlying type of the enumeration (4.9), and subsequently to the enumeration type.
- A prvalue of type "pointer to cv1 B," where B is a class type, can be converted to a prvalue of type "pointer to cv2 D," where D is a class derived (Clause 10) from B, if a valid standard conversion from "pointer to D" to "pointer to B" exists (4.10), cv2 is the same cv-qualification as, or greater cv-qualification than, cv1, and B is neither a virtual base class of D nor a base class of a virtual base class of D. The null pointer value (4.10) is converted to the null pointer value of the destination type. If the prvalue of type "pointer to cv1 B" points

to a B that is actually a subobject of an object of type D, the resulting pointer points to the enclosing object of type D. Otherwise, the 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:

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

— end example]
```

## 5.2.10 Reinterpret cast

## [expr.reinterpret.cast]

- The result of the expression reinterpret\_cast<T>(v) is the result of converting the expression v to type T. If T is an Ivalue reference type or an rvalue reference to function type, the result is an Ivalue; if T is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the Ivalue-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.
- <sup>2</sup> 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.
- <sup>3</sup> [Note: The mapping performed by reinterpret\_cast might, or might not, produce a representation different from the original value. end note]
- A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined. [Note: It is intended to be unsurprising to those who know the addressing structure 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]
- <sup>5</sup> 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]
- A function pointer can be explicitly converted to a function pointer of a different type. The effect of calling a function through a pointer to a function type (8.3.5) that is not the same as the type used in the definition of the function is undefined. Except that converting a prvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are function types) and back to its original type yields the original pointer value, the

§ 5.2.10 101

<sup>69)</sup> Function types (including those used in pointer to member function types) are never cv-qualified; see 8.3.5.

result of such a pointer conversion is unspecified. [ Note: see also 4.10 for more details of pointer conversions. —  $end\ note$  ]

- An object pointer can be explicitly converted to an object pointer of a different type. When a prvalue v of type "pointer to T1" is converted to the type "pointer to cv T2", the result is static\_cast<cv T2\*>(static\_cast<cv void\*>(v)) if both T1 and T2 are standard-layout types (3.9) and the alignment requirements of T2 are no stricter than those of T1, or if either type is void. Converting a prvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value. The result of any other such pointer conversion is unspecified.
- 8 Converting a function pointer to an object pointer type or vice versa is conditionally-supported. The meaning of such a conversion is implementation-defined, except that if an implementation supports conversions in both directions, converting a prvalue of one type to the other type and back, possibly with different cyqualification, shall yield the original pointer value.
- <sup>9</sup> The null pointer value (4.10) is converted to the null pointer value of the destination type. [Note: A null pointer constant of type std::nullptr\_t cannot be converted to a pointer type, and a null pointer constant of integral type is not necessarily converted to a null pointer value. —end note]
- A prvalue of type "pointer to member of X of type T1" can be explicitly converted to a prvalue of a different type "pointer to member of Y of type T2" if T1 and T2 are both function types or both object types.<sup>71</sup> 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 T1" to the type "pointer to data member of Y of type T2" (where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer to member value.
- An Ivalue expression of type T1 can be cast to the type "reference to T2" if an expression of type "pointer to T1" can be explicitly converted to the type "pointer to T2" using a reinterpret\_cast. That is, a reference cast reinterpret\_cast<T&>(x) has the same effect as the conversion \*reinterpret\_cast<T\*>(&x) with the built-in & and \* operators (and similarly for reinterpret\_cast<T&&>(x)). The result refers to the same object as the source Ivalue, but with a different type. The result is an Ivalue for an Ivalue 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.<sup>72</sup>

5.2.11 Const cast [expr.const.cast]

- The result of the expression const\_cast<T>(v) is of type T. If T is an Ivalue reference to object type, the result is an Ivalue; if T is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the Ivalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the expression v. Conversions that can be performed explicitly using const\_cast are listed below. No other conversion shall be performed explicitly using const\_cast.
- [ Note: Subject to the restrictions in this section, an expression may be cast to its own type using a const\_-cast operator. end note ]
- <sup>3</sup> For two pointer types T1 and T2 where

T1 is  $cv_{1,0}$  pointer to  $cv_{1,1}$  pointer to  $\cdots cv_{1,n-1}$  pointer to  $cv_{1,n}$  T

and

<sup>70)</sup> The types may have different *cv*-qualifiers, subject to the overall restriction that a reinterpret\_cast cannot cast away constness.

<sup>71)</sup> T1 and T2 may have different cv-qualifiers, subject to the overall restriction that a reinterpret\_cast cannot cast away constness.

<sup>72)</sup> This is sometimes referred to as a type pun.

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

where T is any object type or the void type and where  $cv_{1,k}$  and  $cv_{2,k}$  may be different cv-qualifications, 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.

- <sup>4</sup> 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 Ivalue of type T1 can be explicitly converted to an Ivalue 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.

- 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.
- A null pointer value (4.10) is converted to the null pointer value of the destination type. The null member pointer value (4.11) is converted to the null member pointer value of the destination type.
- [Note: Depending on the type of the object, a write operation through the pointer, Ivalue or pointer to data member resulting from a const\_cast that casts away a const-qualifier amay 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 is T1cv_{1,1}*\cdots cv_{1,N}* where T1 is not a pointer type X2 is T2cv_{2,1}*\cdots cv_{2,M}* where T2 is not a pointer type K is \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:

```
\mathsf{T} c v_{1,(N-K+1)} * c v_{1,(N-K+2)} * \cdots c v_{1,N} * to \mathsf{T} c v_{2,(M-K+1)} * c v_{2,(M-K+2)} * \cdots c v_{2,M} *
```

- <sup>9</sup> Casting from an Ivalue of type T1 to an Ivalue of type T2 using an Ivalue reference cast or casting from an expression of type T1 to an xvalue of type T2 using an rvalue reference cast casts away constness if a cast from a prvalue of type "pointer to T1" to the type "pointer to T2" casts away constness.
- Casting from a prvalue of type "pointer to data member of X of type T1" to the type "pointer to data member of Y of type T2" casts away constness if a cast from a prvalue of type "pointer to T1" to the type "pointer to T2" casts away constness.

§ 5.2.11 103

<sup>73)</sup>  ${\tt const\_cast}$  is not limited to conversions that cast away a const-qualifier.

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 const cv-qualifier has been cast away.

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

## 5.3 Unary expressions

[expr.unary]

Expressions with unary operators group right-to-left.

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

## \* & + - ! ~ 5.3.1 Unary operators

[expr.unary.op]

- The unary \* operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result is an Ivalue referring to the object or function to which the expression points. If the type of the expression is "pointer to T," the type of the result is "T." [Note: a pointer to an incomplete type (other than cv void) can be dereferenced. The Ivalue thus obtained can be used in limited ways (to initialize a reference, for example); this Ivalue must not be converted to a prvalue, see 4.1. end note]
- <sup>2</sup> The result of each of the following unary operators is a prvalue.
- The result of the unary & operator is a pointer to its operand. The operand shall be an Ivalue or a qualified-id. If the operand is a 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. Otherwise, if the type of the expression is T, the result has type "pointer to T" and is a prvalue that is the address of the designated object (1.7) or a pointer to the designated function. [Note: In particular, the address of an object of type "cv T" is "pointer to cv T", with the same cv-qualification. —end note] [Example:

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

- $end\ example$ ] [Note: a pointer to member formed from a mutable non-static data member (7.1.1) does not reflect the mutable specifier associated with the non-static data member.  $end\ note$ ]
- A pointer to member is only formed when an explicit & is used and its operand is a qualified-id not enclosed in parentheses. [Note: that is, the expression &(qualified-id), where the qualified-id is enclosed in parentheses, does not form an expression of type "pointer to member." Neither does qualified-id, because there is no implicit conversion from a qualified-id for a non-static member function to the type "pointer to member function" as there is from an lvalue of function type to the type "pointer to function" (4.3). Nor is &unqualified-id a pointer to member, even within the scope of the unqualified-id's class. end note]

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

- The address of an overloaded function (Clause 13) can be taken only in a context that uniquely determines which version of the overloaded function is referred to (see 13.4). [Note: since the context might determine whether the operand is a static or non-static member function, the context can also affect whether the expression has type "pointer to function" or "pointer to member function." end note
- <sup>7</sup> 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.
- <sup>8</sup> 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.
- <sup>9</sup> 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]

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

5.3.3 Sizeof [expr.sizeof]

- The sizeof operator yields the number of bytes in the object representation of its operand. The operand is either an expression, which is an unevaluated operand (Clause 5), or a parenthesized type-id. The sizeof operator shall not be applied to an expression that has function or incomplete type, to an enumeration type whose underlying type is not fixed before all its enumerators have been declared, to the parenthesized name of such types, or to an Ivalue that designates a bit-field. sizeof(char), sizeof(signed char) and sizeof(unsigned char) are 1. The result of sizeof applied to any other fundamental type (3.9.1) is implementation-defined. [Note: in particular, sizeof(bool), sizeof(char16\_t), sizeof(char32\_t), and sizeof(wchar\_t) are implementation-defined. end note] [Note: See 1.7 for the definition of byte and 3.9 for the definition of object representation. end note]
- When applied to a reference or a reference type, the result is the size of the referenced type. When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array. The size of a most derived class shall be greater than zero (1.8). The result of applying sizeof to a base class subobject is the size of the base class type. When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of n elements is n times the size of an element.
- <sup>3</sup> The sizeof operator can be applied to a pointer to a function, but shall not be applied directly to a function.

<sup>74)</sup> sizeof(bool) is not required to be 1.

<sup>75)</sup> 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.

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.

The identifier in a sizeof... expression shall name a parameter pack. The sizeof... operator yields the number of arguments provided for the parameter pack *identifier*. A sizeof... expression is a pack expansion (14.5.3). [Example:

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

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

5.3.4 New [expr.new]

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

```
new-expression:
       ::_{opt}new new-placement_{opt} new-type-id new-initializer_{opt}
       ::_{opt}new new-placement_{opt}( type-id ) new-initializer_{opt}
new-placement:
       ( expression-list )
new-type-id:
       type-specifier-seq new-declarator_{opt}
new-declarator:
      ptr-operator new-declarator_{opt}
       noptr-new-declarator
noptr-new-declarator:
       [ expression ] attribute-specifier-seq_{opt}
       noptr-new-declarator [ constant-expression ] attribute-specifier-seq_{opt}
new\mbox{-}initializer:
       ( expression-list_{opt})
       braced	ext{-}init	ext{-}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.

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  $\mathbf{e}$  be the *assignment-expression* in the *new-initializer* and  $\mathbf{T}$  be the *new-type-id* or *type-id* of the *new-expression*, then the allocated type is the type deduced for the variable  $\mathbf{x}$  in the invented declaration (7.1.6.4):

```
— end example]
```

The new-type-id in a new-expression is the longest possible sequence of new-declarators. [Note: this prevents ambiguities between the declarator operators &, &&, \*, and [] and their expression counterparts. —end note] [Example:

```
new int * i; // syntax error: parsed as (new int*) i, not as (new int)*i
```

The \* is the pointer declarator and not the multiplication operator. — end example]

<sup>4</sup> [Note: parentheses in a new-type-id of a new-expression can have surprising effects. [Example:

```
new int(*[10])();  // error
is ill-formed because the binding is
(new int) (*[10])();  // error
```

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

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

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

- When the allocated object is an array (that is, the noptr-new-declarator syntax is used or the new-type-id or type-id denotes an array type), the new-expression yields a pointer to the initial element (if any) of the array. [Note: both new int and new int[10] have type int\* and the type of new int[i] [10] is int (\*) [10] end note] The attribute-specifier-seq in a noptr-new-declarator appertains to the associated array type.
- 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. [Example: given the definition int n = 42, new float[n][5] is well-formed (because n is the expression) of a noptr-new-declarator), but new float[5][n] is ill-formed (because n is not a constant expression). end example]
- 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 less than zero or such that the size of the allocated object would exceed the implementation-defined limit, or if the *new-initializer* is a *braced-init-list* for which the number of *initializer-clauses* exceeds the number of elements to initialize, 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).
- 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 new[] and the deallocation function's name is operator delete[]. [Note: an implementation shall provide default definitions for the global allocation functions (3.7.4, 18.6.1.1, 18.6.1.2). A C++ program can provide alternative definitions of these functions (17.6.4.6) and/or class-specific versions (12.5). end note]
- <sup>9</sup> If the *new-expression* begins with a unary:: operator, the allocation function's name is looked up in the global scope. Otherwise, if the allocated type is a class type T or array thereof, the allocation function's name is looked up in the scope of T. If this lookup fails to find the name, or if the allocated type is not a class type, the allocation function's name is looked up in the global scope.
- A new-expression passes the amount of space requested to the allocation function as the first argument of type std::size\_t. That argument shall be no less than the size of the object being created; it may be

greater than the size of the object being created only if the object is an array. For arrays of char and unsigned char, the difference between the result of the new-expression and the address returned by the allocation function shall be an integral multiple of the strictest fundamental alignment requirement (3.11) of any object type whose size is no greater than the size of the array being created. [Note: Because allocation functions are assumed to return pointers to storage that is appropriately aligned for objects of any type with fundamental alignment, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed. —end note]

- The new-placement syntax is used to supply additional arguments to an allocation function. If used, overload resolution is performed on a function call created by assembling an argument list consisting of the amount of space requested (the first argument) and the expressions in the new-placement part of the new-expression (the second and succeeding arguments). The first of these arguments has type std::size\_t and the remaining arguments have the corresponding types of the expressions in the new-placement.
- [Example:
  - new T results in a call of operator new(sizeof(T)),
  - new(2,f) T results in a call of operator new(sizeof(T),2,f),
  - new T[5] results in a call of operator new[](sizeof(T)\*5+x), and
  - new(2,f) T[5] results in a call of operator new[] (sizeof(T)\*5+y,2,f).

Here, x and y are non-negative unspecified values representing array allocation overhead; the result of the new-expression will be offset by this amount from the value returned by operator new[]. This overhead may be applied in all array new-expressions, including those referencing the library function operator new[](std::size\_t, void\*) and other placement allocation functions. The amount of overhead may vary from one invocation of new to another. — end example]

- [Note: unless an allocation function is declared with 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.
- [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]
- <sup>15</sup> 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.
- 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.
- <sup>17</sup> 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).

If any part of the object initialization described above <sup>76</sup> terminates by throwing an exception and a suitable deallocation function can be found, the deallocation function is called to free the memory in which the object was being constructed, after which the exception continues to propagate in the context of the *new-expression*. If no unambiguous matching deallocation function can be found, propagating the exception does not cause the object's memory to be freed. [*Note:* This is appropriate when the called allocation function does not allocate memory; otherwise, it is likely to result in a memory leak. — *end note*]

- If the new-expression begins with a unary:: operator, the deallocation function's name is looked up in the global scope. Otherwise, if the allocated type is a class type T or an array thereof, the deallocation function's name is looked up in the scope of T. If this lookup fails to find the name, or if the allocated type is not a class type or array thereof, the deallocation function's name is looked up in the global scope.
- A declaration of a placement deallocation function matches the declaration of a placement allocation function if it has the same number of parameters and, after parameter transformations (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:

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 [expr.delete]

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

delete-expression:

```
:: optdelete cast-expression
:: optdelete [ ] 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**. 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

<sup>76)</sup> This may include evaluating a new-initializer and/or calling a constructor.

<sup>77)</sup> 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.

<sup>78)</sup> This implies that an object cannot be deleted using a pointer of type void\* because void is not an object type.

this section. In the first alternative (delete object), the value of the operand of delete may be a null pointer value, a pointer to a non-array object created by a previous new-expression, or a pointer to a subobject (1.8) representing a base class of such an object (Clause 10). If not, the behavior is undefined. In the second alternative (delete array), the value of the operand of delete may be a null pointer value or a pointer value that resulted from a previous array new-expression. If not, the behavior is undefined. [Note: this means that the syntax of the delete-expression must match the type of the object allocated by new, not the syntax of the new-expression. —end note] [Note: a pointer to a const type can be the operand of a delete-expression; it is not necessary to cast away the constness (5.2.11) of the pointer expression before it is used as the operand of the delete-expression. —end note]

- 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.
- <sup>4</sup> The *cast-expression* in a *delete-expression* shall be evaluated exactly once.
- <sup>5</sup> 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.
- <sup>6</sup> 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).
- <sup>7</sup> 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 (17.6.4.6), and/or class-specific versions (12.5). end note]
- When the keyword **delete** in a *delete-expression* is preceded by the unary :: operator, the global deallocation function is used to deallocate the storage.
- <sup>10</sup> Access and ambiguity control are done for both the deallocation function and the destructor (12.4, 12.5).

## 5.3.6 Alignof

[expr.alignof]

- An alignof expression yields the alignment requirement of its operand type. The operand shall be a *type-id* representing a complete object type or an array thereof or a reference to a complete object type.
- <sup>2</sup> The result is an integral constant of type std::size\_t.
- When alignof is applied to a reference type, the result shall be the alignment of the referenced type. When alignof is applied to an array type, the result shall be the alignment of the element type.

## 5.3.7 noexcept operator

[expr.unary.noexcept]

The noexcept operator determines whether the evaluation of its operand, which is an unevaluated operand (Clause 5), can throw an exception (15.1).

 $no except\mbox{-}expression:$ 

 ${\tt noexcept}$  (  ${\it expression}$  )

- The result of the noexcept operator is a constant of type bool and is an rvalue.
- The result of the noexcept operator is false if in a potentially-evaluated context the expression would contain
  - a potentially evaluated call<sup>80</sup> to a function, member function, function pointer, or member function pointer that does not have a non-throwing *exception-specification* (15.4), unless the call is a constant expression (5.19),

<sup>79)</sup> 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.

<sup>80)</sup> This includes implicit calls such as the call to an allocation function in a new-expression.

- 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)

[expr.cast]

- The result of the expression (T) cast-expression is of type T. The result is an Ivalue if T is an Ivalue 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.

 $cast\mbox{-}expression:$ 

```
unary-expression
```

( type-id ) cast-expression

- Any type conversion not mentioned below and not explicitly defined by the user (12.3) is ill-formed.
- <sup>4</sup> 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.

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:

§ 5.4 111

```
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]

The pointer-to-member operators ->\* and .\* group left-to-right.

```
pm-expression:
    cast-expression
    pm-expression .* cast-expression
    pm-expression ->* cast-expression
```

- The binary operator .\* binds its second operand, which shall be of type "pointer to member of T" (where T is a completely-defined class type) to its first operand, which shall be of class T or of a class of which T is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.
- The binary operator ->\* 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 expression E1->\*E2 is converted into the equivalent form (\*(E1)).\*E2.
- 4 Abbreviating *pm-expression.*\**cast-expression* as E1.\*E2, E1 is called the *object expression*. If the dynamic type of E1 does not contain the member to which E2 refers, the behavior is undefined.
- 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,

<sup>6</sup> 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 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. If the second operand is the null pointer to member value (4.11), the behavior is undefined.

## 5.6 Multiplicative operators

[expr.mul]

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

```
multiplicative-expression:

pm-expression

multiplicative-expression * pm-expression

multiplicative-expression / pm-expression

multiplicative-expression % pm-expression
```

- <sup>2</sup> 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.
- <sup>3</sup> The binary \* operator indicates multiplication.
- <sup>4</sup> 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;<sup>81</sup> 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]

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.

- <sup>2</sup> 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.
- <sup>3</sup> 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.
- <sup>4</sup> 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.
- 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

§ 5.7 113

<sup>81)</sup> This is often called truncation towards zero.

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.

- When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The type of the result is an implementation-defined signed integral type; this type shall be the same type that is defined as  $std::ptrdiff_t$  in the <cstddef> header (18.2). As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined. In other words, if the expressions P and Q point to, respectively, the i-th and j-th elements of an array object, the expression (P)-(Q) has the value i-j provided the value fits in an object of type  $std::ptrdiff_t$ . Moreover, if the expression P points either to an element of an array object or one past the last element of an array object, and the expression Q points to the last element of the same array object, the expression ((Q)+1)-(P) has the same value as ((Q)-(P))+1 and as -((P)-((Q)+1)), and has the value zero if the expression P points one past the last element of the array object, even though the expression (Q)+1 does not point to an element of the array object. Unless both pointers point to elements of the same array object, or one past the last element of the array object, the behavior is undefined.<sup>82</sup>
- 7 If the value 0 is added to or subtracted from a pointer value, the result compares equal to the original pointer value. If two pointers point to the same object or both point one past the end of the same array or both are null, and the two pointers are subtracted, the result compares equal to the value 0 converted to the type std::ptrdiff\_t.

5.8 Shift operators [expr.shift]

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.

- 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.
- <sup>3</sup> 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

[expr.rel]

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

§ 5.9 114

<sup>82)</sup> 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.

```
— 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.

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:

- $-end\ example$ ] 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.

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.
- <sup>5</sup> 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

[expr.eq]

```
equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression
```

- The == (equal to) and the != (not equal to) operators have the same semantic restrictions, conversions, and result type as the relational operators except for their lower precedence and truth-value result. [Note: a<br/>b == c<d is true whenever a<br/>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).
- In addition, pointers to member can be compared, or a pointer to member and a null pointer constant. Pointer to member conversions (4.11) and qualification conversions (4.4) are performed to bring them to a common type. If one operand is a null pointer constant, the common type is the type of the other operand. Otherwise, the common type is a pointer to member type similar (4.4) to the type of one of the operands, with a cv-qualification signature (4.4) that is the union of the cv-qualification signatures of the operand types. [Note: this implies that any pointer to member can be compared to a null pointer constant. end note] If both operands are null, they compare equal. Otherwise if only one is null, they compare unequal. Otherwise if either is a pointer to a virtual member function, the result is unspecified. Otherwise they compare equal if and only if they would refer to the same member of the same most derived object (1.8) or the same subobject if they were dereferenced with a hypothetical object of the associated class type. [Example:

```
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]
```

- <sup>3</sup> If two operands of type std::nullptr\_t are compared, the result is true if the operator is ==, and false otherwise.
- <sup>4</sup> Each of the operators shall yield true if the specified relationship is true and false if it is false.

## 5.11 Bitwise AND operator

[expr.bit.and]

```
and-expression:
equality-expression
and-expression & equality-expression
```

The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral or unscoped enumeration operands.

## 5.12 Bitwise exclusive OR operator

[expr.xor]

exclusive-or-expression:

and-expression

exclusive-or-expression ^ and-expression

The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral or unscoped enumeration operands.

## 5.13 Bitwise inclusive OR operator

[expr.or]

 $inclusive ext{-} or ext{-} expression:$ 

exclusive - or - expression

inclusive-or-expression | exclusive-or-expression

<sup>1</sup> The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral or unscoped enumeration operands.

## 5.14 Logical AND operator

[expr.log.and]

logical-and-expression:

 $inclusive \hbox{-} or \hbox{-} expression$ 

 $logical\text{-} and\text{-} expression \ \&\& \ inclusive\text{-} or\text{-} expression$ 

- The && operator groups left-to-right. The operands are both contextually converted to type bool (Clause 4). The result is true if both operands are true and false otherwise. Unlike &, && guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is false.
- <sup>2</sup> 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

[expr.log.or]

 $logical \hbox{-} or \hbox{-} expression:$ 

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.
- <sup>2</sup> 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

[expr.cond]

 $conditional\mbox{-}expression:$ 

logical-or-expression

 $logical \hbox{-} or \hbox{-} expression ~?~ expression ~:~ assignment \hbox{-} expression$ 

- 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.
- <sup>2</sup> If either the second or the third operand has type 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 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 E1 and E2 have class type, and the underlying class types are the same or one is a base class of the other: E1 can be converted to match E2 if the class of T2 is the same type as, or a base class of, the class of T1, and the cv-qualification of T2 is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of T1. If the conversion is applied, E1 is changed to a prvalue of type T2 by copy-initializing a temporary of type T2 from E1 and using that temporary as the converted operand.
    - Otherwise (i.e., if E1 or E2 has a nonclass type, or if they both have class types but the underlying classes are not either the same or one a base class of the other): E1 can be converted to match E2 if E1 can be implicitly converted to the type that expression E2 would have if E2 were converted to a prvalue (or the type it has, if 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.

- <sup>4</sup> 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.
- <sup>5</sup> 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.
- <sup>6</sup> 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

[expr.ass]

The assignment operator (=) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand and return an lvalue referring to the left operand. The result in all cases is a bit-field if the left operand is a bit-field. In all cases, the assignment is sequenced after the value computation of the right and left operands, and before the value computation of the assignment expression. With respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. [Note: Therefore, a function call shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single compound assignment operator. — end note]

```
assignment-expression:
    conditional-expression
    logical-or-expression assignment-operator initializer-clause
    throw-expression

assignment-operator: one of

= *= /= %= += -= >>= <<= &= ^= |=
```

- In simple assignment (=), the value of the expression replaces that of the object referred to by the left operand.
- <sup>3</sup> 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.
- <sup>4</sup> 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).
- <sup>5</sup> [Note: For class objects, assignment is not in general the same as initialization (8.5, 12.1, 12.6, 12.8). end note]
- When the left operand of an assignment operator denotes a reference to T, the operation assigns to the object of type T denoted by the reference.
- <sup>7</sup> The behavior of an expression of the form E1 op = E2 is equivalent to E1 = E1 op E2 except that E1 is evaluated only once. In += and -=, E1 shall either have arithmetic type or be a pointer to a possibly cv-qualified completely-defined object type. In all other cases, E1 shall have arithmetic type.
- If the value being stored in an object is accessed from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have the same type, otherwise the behavior is undefined. [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.

## 5.18 Comma operator

[expr.comma]

The comma operator groups left-to-right.

expression:

 $assignment\hbox{-} expression$ 

expression , assignment-expression

A pair of expressions separated by a comma is evaluated left-to-right; the left expression is a discarded-value expression (Clause 5).<sup>83</sup> 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.

<sup>2</sup> 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

[expr.const]

Certain contexts require expressions that satisfy additional requirements as detailed in this sub-clause; other contexts have different semantics depending on whether or not an expression satisfies these requirements. Expressions that satisfy these requirements are called *constant expressions*. [Note: Constant expressions can be evaluated during translation. — end note]

 $constant\mbox{-}expression:$ 

 $conditional\hbox{-} expression$ 

- A conditional-expression is a core 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 constructor for a literal class or a constexpr function [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 substituted by function invocation substitution (7.1.5), do not produce a constant expression; [Example:

<sup>83)</sup> However, an invocation of an overloaded comma operator is an ordinary function call; hence, the evaluations of its argument expressions are unsequenced relative to one another (see 1.9).

— an invocation of a **constexpr** constructor with arguments that, when substituted by function invocation substitution (7.1.5), do not produce all constant expressions for the constructor calls and full-expressions in the *mem-initializers*; [Example:

- an invocation of a constexpr function or a constexpr constructor that would exceed the implementation-defined recursion limits (see Annex B);
- 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 whose lifetime has not ended, initialized with a constant expression;
- an lvalue-to-rvalue conversion (4.1) that is applied to a glvalue that refers to a non-active member of a union or a subobject thereof;
- an *id-expression* that refers to a variable or data member of reference type unless the reference has a preceding initialization, initialized with a constant expression;

```
a dynamic cast (5.2.7);
a reinterpret_cast (5.2.10);
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;
```

§ 5.19 121

```
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 the result is unspecified;
an assignment or a compound assignment (5.17); or
a throw-expression (15.1).
```

- A literal constant expression is a prvalue core constant expression of literal type, but not pointer type. An integral constant expression is a literal constant expression of integral or unscoped enumeration type. [Note: Such expressions may be used as array bounds (8.3.4, 5.3.4), as bit-field lengths (9.6), as enumerator initializers if the underlying type is not fixed (7.2), as null pointer constants (4.10), and as alignments (7.6.2). —end note] A converted constant expression of type T is a literal constant expression, implicitly converted to type T, where the implicit conversion (if any) is permitted in a literal constant expression and the implicit conversion sequence contains only user-defined conversions, lvalue-to-rvalue conversions (4.1), integral promotions (4.5), and integral conversions (4.7) other than narrowing conversions (8.5.4). [Note: such expressions may be used as case expressions (6.4.2), as enumerator initializers if the underlying type is fixed (7.2), and as integral or enumeration non-type template arguments (14.3). —end note] A reference constant expression is an Ivalue core constant expression that designates an object with static storage duration or a function. An address constant expression is a prvalue core constant expression of pointer type that evaluates to the address of an object with static storage duration, to the address of a function, or to a null pointer value, or a prvalue core constant expression of type std::nullptr\_t. Collectively, literal constant expressions, reference constant expressions, and address constant expressions are called constant expressions.
- <sup>4</sup> [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 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.<sup>84</sup> [Example:

```
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 unscoped enumeration type and that conversion function shall be constexpr. [Example:

```
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;
```

<sup>84)</sup> Nonetheless, implementations are encouraged to provide consistent results, irrespective of whether the evaluation was actually performed during translation or during program execution.

 $\odot$  ISO/IEC N3337

```
X<a> x;  // OK: unique conversion to int
int ary[a];  // error: ambiguous conversion

— end example]
```

## 6 Statements

# [stmt.stmt]

Except as indicated, statements are executed in sequence.

```
statement: \\ labeled-statement \\ attribute-specifier-seq_{opt} \ expression-statement \\ attribute-specifier-seq_{opt} \ compound-statement \\ attribute-specifier-seq_{opt} \ selection-statement \\ attribute-specifier-seq_{opt} \ iteration-statement \\ attribute-specifier-seq_{opt} \ jump-statement \\ declaration-statement \\ declaration-statement \\ \end{cases}
```

attribute-specifier-seq $_{opt}$  try-block The optional attribute-specifier-seq appertains to the respective statement.

## 6.1 Labeled statement

[stmt.label]

A statement can be labeled.

```
\begin{tabular}{ll} labeled-statement:\\ attribute-specifier-seq_{opt}\ identifier: statement\\ attribute-specifier-seq_{opt} {\tt case}\ constant-expression: statement\\ attribute-specifier-seq_{opt} {\tt default}: statement\\ \end{tabular}
```

The optional attribute-specifier-seq appertains to the label. An identifier label declares the identifier. The only use of an identifier label is as the target of a goto. The scope of a label is the function in which it appears. Labels shall not be redeclared within a function. A label can be used in a goto statement before its definition. Labels have their own name space and do not interfere with other identifiers.

<sup>2</sup> Case labels and default labels shall occur only in switch statements.

## 6.2 Expression statement

[stmt.expr]

Expression statements have the form

```
expression-statement: expression_{opt};
```

The expression is a discarded-value expression (Clause 5). All side effects from an expression statement are completed before the next statement is executed. An expression statement with the expression missing is called a null statement. [Note: Most statements are expression statements — usually assignments or function calls. A null statement is useful to carry a label just before the  $\}$  of a compound statement and to supply a null body to an iteration statement such as a while statement (6.5.1). — end note]

## 6.3 Compound statement or block

[stmt.block]

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided.

```
compound-statement: \\ \{statement-seq_{opt}\} \\ statement-seq: \\ statement \\ statement-seq statement
```

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

§ 6.3

note

#### 6.4 Selection statements

[stmt.select]

Selection statements choose one of several flows of control.

```
selection\text{-}statement: \\ \text{if (condition) statement} \\ \text{if (condition) statement else statement} \\ \text{switch (condition) statement} \\ condition: \\ expression \\ attribute\text{-}specifier\text{-}seq_{opt} decl\text{-}specifier\text{-}seq declarator = initializer\text{-}clause} \\ attribute\text{-}specifier\text{-}seq_{opt} decl\text{-}specifier\text{-}seq declarator braced\text{-}init\text{-}list} \\ \end{cases}
```

See 8.3 for the optional attribute-specifier-seq in a condition. In Clause 6, the term substatement refers to the contained statement or statements that appear in the syntax notation. The substatement in a selection-statement (each substatement, in the else form of the if statement) implicitly defines a block scope (3.3). If the substatement in a selection-statement is a single statement and not a compound-statement, it is as if it was rewritten to be a compound-statement containing the original substatement. [Example:

```
if (x)
  int i;
  can be equivalently rewritten as
if (x) {
  int i;
}
```

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

- 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.
- 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:

— end example]

- 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.
- <sup>5</sup> 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.

§ 6.4 125

<sup>6</sup> In the decl-specifier-seq of a condition, each decl-specifier shall be either a type-specifier or constexpr.

#### 6.4.1 The if statement

[stmt.if]

<sup>1</sup> 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.<sup>85</sup>

#### 6.4.2 The switch statement

[stmt.switch]

- The switch statement causes control to be transferred to one of several statements depending on the value of a condition.
- The condition shall be of integral type, enumeration type, or of a class type for which a single 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 a converted constant expression (5.19) of the promoted type of the switch condition. No two of the case constants in the same switch shall have the same value after conversion to the promoted type of the switch condition.

There shall be at most one label of the form

#### default :

within a switch statement.

- <sup>4</sup> Switch statements can be nested; a case or default label is associated with the smallest switch enclosing it.
- When the switch statement is executed, its condition is evaluated and compared with each case constant. If one of the case constants is equal to the value of the condition, control is passed to the statement following the matched case label. If no case constant matches the condition, and if there is a default label, control passes to the statement labeled by the default label. If no case matches and if there is no default then none of the statements in the switch is executed.
- case and default labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see break, 6.6.1. [Note: Usually, the substatement that is the subject of a switch is compound and case and default labels appear on the top-level statements contained within the (compound) substatement, but this is not required. Declarations can appear in the substatement of a switch-statement. end note]

#### 6.5 Iteration statements

[stmt.iter]

Iteration statements specify looping.

```
iteration\mbox{-}statement:
```

```
while ( condition ) statement do statement while ( expression ); for ( for\text{-}init\text{-}statement condition_{opt}; expression_{opt}) statement for ( for\text{-}range\text{-}declaration : for\text{-}range\text{-}initializer ) statement for\text{-}init\text{-}statement: expression\text{-}statement simple\text{-}declaration for\text{-}range\text{-}declaration: attribute\text{-}specifier\text{-}seq_{opt} decl\text{-}specifier\text{-}seq declarator for\text{-}range\text{-}initializer: expression braced\text{-}init\text{-}list
```

§ 6.5

<sup>85)</sup> In other words, the else is associated with the nearest un-elsed if.

See 8.3 for the optional attribute-specifier-seq in a for-range-declaration. [Note: A for-init-statement ends with a semicolon. — end note]

The substatement in an *iteration-statement* implicitly defines a block scope (3.3) which is entered and exited each time through the loop.

If the substatement in an iteration-statement is a single statement and not a *compound-statement*, it is as if it was rewritten to be a compound-statement containing the original statement. [Example:

```
while (--x >= 0)
  int i;
  can be equivalently rewritten as
while (--x >= 0) {
  int i;
}
```

- Thus after the while statement, i is no longer in scope. end example
- <sup>4</sup> [Note: The requirements on conditions in iteration statements are described in 6.4. end note]

## 6.5.1 The while statement

[stmt.while]

- <sup>1</sup> 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.
- 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

The variable created in a condition is destroyed and created with each iteration of the loop. [Example:

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

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

#### 6.5.2 The do statement

[stmt.do]

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

§ 6.5.2

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.

#### The for statement 6.5.3

[stmt.for]

The for statement

```
for (for-init-statement condition<sub>opt</sub>; expression<sub>opt</sub>) 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

- <sup>2</sup> Either or both of the condition and the expression can be omitted. A missing condition makes the implied while Clause equivalent to while (true).
- <sup>3</sup> If the for-init-statement is a declaration, the scope of the name(s) declared extends to the end of the forstatement. [Example:

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

## The range-based for statement

[stmt.ranged]

For a range-based for statement of the form

```
for (for-range-declaration: expression) statement
   let range-init be equivalent to the expression surrounded by parentheses<sup>86</sup>
         ( expression )
   and for a range-based for statement of the form
         for (for-range-declaration: braced-init-list) statement
   let range-init be equivalent to the braced-init-list. In each case, a range-based for statement is equivalent
to
    auto && __range = range-init;
    for ( auto __begin = begin-expr,
                __end = end-expr;
           __begin != __end;
           ++__begin ) {
      for-range-declaration = *_begin;
```

§ 6.5.4 128

<sup>86)</sup> this ensures that a top-level comma operator cannot be reinterpreted as a delimiter between init-declarators in the declaration of \_\_range.

```
}
}
```

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;
- if \_RangeT is a class type, the *unqualified-ids* begin and end are looked up in the scope of class \_RangeT as if by class member access lookup (3.4.5), and if either (or both) finds at least one declaration, *begin-expr* and *end-expr* are \_\_range.begin() and \_\_range.end(), respectively;
- 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:
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
x *= 2;
```

— end example]

<sup>2</sup> In the decl-specifier-seq of a for-range-declaration, each decl-specifier shall be either a type-specifier or constexpr.

## 6.6 Jump statements

[stmt.jump]

Jump statements unconditionally transfer control.

```
jump-statement:
    break ;
    continue ;
    return expression<sub>opt</sub>;
    return braced-init-list ;
    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 std::exit() or std::abort() (18.5), for example) without destroying class objects with automatic storage duration.—end note]

#### 6.6.1 The break statement

[stmt.break]

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

## 6.6.2 The continue statement

[stmt.cont]

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

§ 6.6.2

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

## 6.6.3 The return statement

[stmt.return]

- A function returns to its caller by the return statement.
- A return statement with neither an expression nor a braced-init-list can be used only in functions that do not return a value, that is, a function with the return type void, a constructor (12.1), or a destructor (12.4). A return statement with an expression of non-void type can be used only in functions returning a value; the value of the expression is returned to the caller of the function. The 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 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:

```
std::pair<std::string,int> f(const char* p, int x) {
  return {p,x};
}
```

— end example]

Flowing off the end of a function is equivalent to a **return** with no value; this results in undefined behavior in a value-returning function.

<sup>3</sup> A return statement with an expression of type 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

[stmt.goto]

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

[stmt.dcl]

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

 $declaration\mbox{-}statement:$ 

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.

- <sup>2</sup> Variables with automatic storage duration (3.7.3) are initialized each time their *declaration-statement* is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).
- It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps<sup>87</sup> 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:

```
void f() {
   // ...
  goto lx;   // ill-formed: jump into scope of a
```

§ 6.7

<sup>87)</sup> The transfer from the condition of a switch statement to a case label is considered a jump in this respect.

```
// ...
ly:
    X a = 1;
    // ...
lx:
    goto ly;
    // OK, jump implies destructor
    // call for a followed by construction
    // again immediately following label ly
}
```

The zero-initialization (8.5) of all block-scope variables with static storage duration (3.7.1) or thread storage duration (3.7.2) is performed before any other initialization takes place. Constant initialization (3.6.2) of a block-scope entity with static storage duration, if applicable, is performed before its block is first entered. An implementation is permitted to perform early initialization of other block-scope variables with static or thread storage duration under the same conditions that an implementation is permitted to statically initialize a variable with static or thread storage duration in namespace scope (3.6.2). Otherwise such a variable is initialized the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization. [Example:

— end example]

- end example]

<sup>5</sup> 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

[stmt.ambig]

There is an ambiguity in the grammar involving expression-statements and declarations: An expression-statement with a function-style explicit type conversion (5.2.3) as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a (. In those cases the statement is a declaration. [Note: To disambiguate, the whole statement might have to be examined to determine if it is an expression-statement or a declaration. This disambiguates many examples. [Example: assuming T is a simple-type-specifier (7.1.6),

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]

<sup>2</sup> The remaining cases are declarations. [Example:

§ 6.8

<sup>88)</sup> The implementation must not introduce any deadlock around execution of the initializer.

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

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

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

§ 6.8

# 7 Declarations

[dcl.dcl]

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

```
declaration-seq:
       declaration
       declaration\hbox{-}seq\ declaration
declaration:
       block\text{-}declaration
       function-definition
       template-declaration
       explicit	ext{-}instantiation
       explicit\text{-}specialization
       linkage-specification
       namespace-definition
       empty-declaration
       attribute-declaration
block-declaration:
       simple-declaration
       asm-definition
       name space-alias-definition
       using-declaration
       using-directive
       static\_assert\text{-}declaration
       a lias\text{-}declaration
       opaque\text{-}enum\text{-}declaration
alias-declaration:
      using identifier\ attribute-specifier-seq_{opt} = type-id;
simple-declaration:
       decl-specifier-seq_{opt} init-declarator-list_{opt};
       attribute-specifier-seq decl-specifier-seq_{opt} init-declarator-list;
static assert-declaration:
       static_assert ( constant-expression , string-literal ) ;
empty\mbox{-}declaration:
attribute-declaration:
       attribute-specifier-seq;
```

[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

attribute-specifier-seq $_{opt}$  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 attribute-specifier-seq in a simple-declaration appertains to each of the entities declared by the declarators of the init-declarator-list. [Note: In the declaration for an entity, attributes appertaining to that entity may appear at the start of the declaration and after the declarator-id for that declaration. — end note] [Example:

```
[[noreturn]] void f [[noreturn]] (); // OK
```

Declarations 133

```
— end example]
```

Except where otherwise specified, the meaning of an attribute-declaration is implementation-defined.

A declaration occurs in a scope (3.3); the scope rules are summarized in 3.4. A declaration that declares a function or defines a class, namespace, template, or function also has one or more scopes nested within it. These nested scopes, in turn, can have declarations nested within them. Unless otherwise stated, utterances in Clause 7 about components in, of, or contained by a declaration or subcomponent thereof refer only to those components of the declaration that are *not* nested within scopes nested within the declaration.

In a simple-declaration, the optional init-declarator-list can be omitted only when declaring a class (Clause 9) or enumeration (7.2), that is, when the decl-specifier-seq contains either a class-specifier, an elaborated-type-specifier with a class-key (9.1), or an enum-specifier. In these cases and whenever a class-specifier or enum-specifier is present in the decl-specifier-seq, the identifiers in these specifiers are among the names being declared by the declaration (as class-names, enum-names, or enumerators, depending on the syntax). In such cases, and except for the declaration of an unnamed bit-field (9.6), the decl-specifier-seq shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration. [Example:

— end example]

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]
```

- <sup>5</sup> An *empty-declaration* has no effect.
- <sup>6</sup> 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*.
- <sup>7</sup> 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 declaration to make a *function-definition*. An object declaration, however, is also a definition unless it contains the **extern** specifier and has no initializer (3.1). A definition causes the appropriate amount of storage to be reserved and any appropriate initialization (8.5) to be done.
- <sup>9</sup> Only in function declarations for constructors, destructors, and type conversions can the *decl-specifier-seq* be omitted.<sup>89</sup>

7.1 Specifiers [dcl.spec]

The specifiers that can be used in a declaration are

decl-specifier:
 storage-class-specifier
 type-specifier
 function-specifier
 friend
 typedef
 constexpr

<sup>89)</sup> The "implicit int" rule of C is no longer supported.

```
decl-specifier-seq:

decl-specifier attribute-specifier-seq_{opt}

decl-specifier decl-specifier-seq
```

The optional attribute-specifier-seq in a decl-specifier-seq appertains to the type determined by the preceding decl-specifiers (8.3). The attribute-specifier-seq affects the type only for the declaration it appears in, not other declarations involving the same type.

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:

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

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

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:

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

— end example] — end note]
```

## 7.1.1 Storage class specifiers

[dcl.stc]

The storage class specifiers are

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

At most one storage-class-specifier shall appear in a given decl-specifier-seq, except that thread\_local may appear with static or extern. If thread\_local appears in any declaration of a variable it shall be present in all declarations of that entity. If a storage-class-specifier appears in a decl-specifier-seq, there can be no typedef specifier in the same decl-specifier-seq and the init-declarator-list 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.

- 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.
- <sup>3</sup> A register specifier is a hint to the implementation that the variable so declared will be heavily used. [Note: The hint can be ignored and in most implementations it will be ignored if the address of the variable is taken. This use is deprecated (see D.2). —end note]
- <sup>4</sup> 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.

- 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.
- 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]
- 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.
- <sup>8</sup> 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:

```
// f() has internal linkage
 static char* f();
 char* f()
                                      // f() still has internal linkage
   { /* ... */ }
 char* g();
                                      // g() has external linkage
 static char* g()
                                      // error: inconsistent linkage
   { /* ... */ }
 void h();
                                      // external linkage
 inline void h();
 inline void 1();
                                      // external linkage
 void 1();
 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
                                      // b has internal linkage
 static int b;
                                      // b still has internal linkage
 extern int b;
                                      // c has external linkage
 int c;
                                      // error: inconsistent linkage
 static int c;
                                      // d has external linkage
 extern int d;
                                      // error: inconsistent linkage
 static int d;
— end example]
```

<sup>9</sup> 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:

— 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:

— 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

[dcl.fct.spec]

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

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

- <sup>2</sup> A function declaration (8.3.5, 9.3, 11.3) with an inline specifier declares an *inline function*. The inline specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism. An implementation is not required to perform this inline substitution at the point of call; however, even if this inline substitution is omitted, the other rules for inline functions defined by 7.1.2 shall still be respected.
- <sup>3</sup> A function defined within a class definition is an inline function. The inline specifier shall not appear on a block scope function declaration.<sup>90</sup> If the inline specifier is used in a friend declaration, that declaration shall be a definition or the function shall have previously been declared inline.
- An inline function shall be defined in every translation unit in which it is odr-used and shall have exactly the same definition in every case (3.2). [Note: A call to the inline function may be encountered before its definition appears in the translation unit. end note] If the definition of a function appears in a translation unit before its first declaration as inline, the program is ill-formed. If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required. An inline function with external linkage shall have the same address in all translation units. A static local variable in an extern inline function always refers to the same object. A 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 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.
- <sup>5</sup> The virtual specifier shall be used only in the initial declaration of a non-static class member function; see 10.3.

90) The inline keyword has no effect on the linkage of a function.

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.

## The typedef specifier

[dcl.typedef]

Declarations containing the decl-specifier typedef declare identifiers that can be used later for naming fundamental (3.9.1) or compound (3.9.2) types. The typedef specifier shall not be combined in a declspecifier-seq with any other kind of specifier except a type-specifier, and it shall not be used in the declspecifier-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;
extern KLICKSP metricp;
```

are all correct declarations; the type of distance is int and that of metricp is "pointer to int." — end

A typedef-name can also be introduced by an alias-declaration. The identifier following the using keyword becomes a typedef-name and the optional attribute-specifier-seq following the identifier appertains to that typedef-name. It has the same semantics as if it were introduced by the typedef specifier. In particular, it does not define a new type and it shall not appear in the type-id. [Example:

```
using handler_t = void (*)(int);
extern handler_t ignore;
                                     // redeclare ignore
extern void (*ignore)(int);
using cell = pair<void*, cell*>;
                                     // ill-formed
  - end example]
```

In a given non-class scope, a typedef specifier can be used to redefine the name of any type declared in that scope to refer to the type to which it already refers. [Example:

```
typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;
```

— end example]

<sup>4</sup> 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
                                // OK
  typedef struct B B;
                                // error
  typedef A A;
};
```

- end example]

<sup>5</sup> If a typedef specifier is used to redefine in a given scope an entity that can be referenced using an elaboratedtype-specifier, the entity can continue to be referenced by an elaborated-type-specifier or as an enumeration or class name in an enumeration or class definition respectively. [Example:

```
struct S;
typedef struct S S;
int main() {
   struct S* p;  // OK
}
struct S { };  // OK
```

— end example]

<sup>6</sup> 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:

<sup>7</sup> Similarly, in a given scope, a class or enumeration shall not be declared with the same name as a *typedef-name* that is declared in that scope and refers to a type other than the class or enumeration itself. [Example:

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

— end example]

[Note: A typedef-name that names a class type, or a cv-qualified version thereof, is also a class-name (9.1). If a typedef-name is used to identify the subject of an elaborated-type-specifier (7.1.6.3), a class definition (Clause 9), a constructor declaration (12.1), or a destructor declaration (12.4), the program is ill-formed.—end note] [Example:

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

typedef struct S T;

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

— end example]

<sup>9</sup> 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 11.3.

#### 7.1.5 The constexpr specifier

[dcl.constexpr]

The constexpr specifier shall be applied only to the definition of a variable, 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
```

```
// error: pixel is a type
 constexpr struct pixel {
   int x:
   int y;
   constexpr pixel(int);
                                    // OK: declaration
 };
 constexpr pixel::pixel(int a)
                                    // OK: definition
   : x(square(a)), y(square(a))
   { }
 constexpr pixel small(2);
                                    // error: square not defined, so small(2)
                                     // not constant (5.19) so constexpr not satisfied
                                    // OK: definition
 constexpr int square(int x) {
   return x * x;
                                    // OK: square defined
 constexpr pixel large(4);
 int next(constexpr int x) {
                                    // error: not for parameters
       return x + 1;
 }
 extern constexpr int memsz;
                                    // error: not a definition
-- \, end \, \, example \, ]
```

- <sup>2</sup> 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;
  - each of its parameter types shall be a literal type;
  - its function-body shall be = delete, = default, or a compound-statement that contains only
    - null statements,
    - static\_assert-declarations
    - typedef declarations and alias-declarations that do not define classes or enumerations,
    - using-declarations,
    - using-directives,
    - and exactly one return statement;
  - every constructor call and implicit conversion used in initializing the return value (6.6.3, 8.5) shall be one of those allowed in a constant expression (5.19).

## [Example:

In a definition of a constexpr constructor, each of the parameter types shall be a literal type. In addition, either its function-body shall be = delete or = default or it shall satisfy the following constraints:

- the class shall not have any virtual base classes;
- its function-body shall not be a function-try-block;
- the *compound-statement* of its *function-body* shall contain only
  - null statements,

— end example]

- static\_assert-declarations
- typedef declarations and alias-declarations that do not define classes or enumerations,
- using-declarations,
- and using-directives;
- every non-static data member and base class sub-object shall be initialized (12.6.2);
- every constructor involved in initializing non-static data members and base class sub-objects shall be a constexpr constructor;
- every assignment-expression that is an initializer-clause appearing directly or indirectly within a brace-or-equal-initializer for a non-static data member that is not named by a mem-initializer-id shall be a constant expression; and
- 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.

```
[Example:
struct Length {
  explicit constexpr Length(int i = 0) : val(i) { }
private:
    int val;
};
— end example]
```

Function invocation substitution for a call of a constexpr function or of a constexpr constructor means implicitly converting each argument to the corresponding parameter type as if by copy-initialization, <sup>91</sup> substituting that converted expression for each use of the corresponding parameter in the function-body, and, for constexpr functions, implicitly converting the resulting returned expression or braced-init-list to the return type of the function as if by copy-initialization. Such substitution does not change the meaning. [Example:

<sup>91)</sup> The resulting converted value will include an lvalue-to-rvalue conversion (4.1) if the corresponding copy-initialization requires one.

For a constexpr function, if no function argument values exist such that the function invocation substitution would produce a constant expression (5.19), the program is ill-formed; no diagnostic required. For a constexpr constructor, if no argument values exist such that after function invocation substitution, every constructor call and full-expression in the *mem-initializers* would be a constant expression (including conversions), the program is ill-formed; no diagnostic required. [Example:

- end example]
- If the instantiated template specialization of a constexpr function template or member function of a class template would fail to satisfy the requirements for a constexpr function or constexpr constructor, that specialization is not a constexpr function or constexpr constructor. [Note: If the function is a member function it will still be const as described below. end note] If no specialization of the template would yield a constexpr function or constexpr constructor, the program is ill-formed; no diagnostic required.
- <sup>7</sup> 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 keyword const is ignored if it appears in the cv-qualifier-seq of the function declarator of the declaration of such a member function. The class of which that function is a member shall be a literal type (3.9). [Example:

```
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 have literal type and shall be initialized. If it is initialized by a constructor call, that call shall be a constant expression (5.19). Otherwise, or if a constexpr specifier is used in a reference declaration, every full-expression that appears in its initializer shall be a constant expression. 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:

```
struct pixel {
   int x, y;
};
constexpr pixel ur = { 1294, 1024 };// OK
constexpr pixel origin; // error: initializer missing
— end example]
```

## 7.1.6 Type specifiers

[dcl.type]

The type-specifiers are

```
type-specifier:
    trailing-type-specifier
    class-specifier
    enum-specifier

trailing-type-specifier:
    simple-type-specifier
    elaborated-type-specifier
    typename-specifier
    cv-qualifier

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

trailing-type-specifier-seq:
    trailing-type-specifier attribute-specifier-seqopt
    trailing-type-specifier trailing-type-specifier-seq
```

The optional attribute-specifier-seq in a type-specifier-seq or a trailing-type-specifier-seq appertains to the type denoted by the preceding type-specifiers (8.3). The attribute-specifier-seq affects the type only for the declaration it appears in, not other declarations involving the same type.

- As a general rule, at most one type-specifier is allowed in the complete decl-specifier-seq of a declaration or in a type-specifier-seq or trailing-type-specifier-seq. The only exceptions to this rule are the following:
  - 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.
- <sup>3</sup> 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. <sup>92</sup> A *type-specifier-seq* shall not define a class or enumeration unless it appears in the *type-id* of an *alias-declaration* (7.1.3) that is not the *declaration* of a *template-declaration*.
- <sup>4</sup> [Note: enum-specifiers, class-specifiers, and typename-specifiers are discussed in 7.2, 9, and 14.6, respectively. The remaining type-specifiers are discussed in the rest of this section. end note]

#### 7.1.6.1 The cv-qualifiers

[dcl.type.cv]

- There are two cv-qualifiers, const and volatile. If a cv-qualifier appears in a decl-specifier-seq, the init-declarator-list of the declaration shall not be empty. [Note: 3.9.3 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 const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path. [Note: Cv-qualifiers are supported by the type system so that they cannot be subverted without casting (5.2.11). end note]
- <sup>4</sup> 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:

```
const int ci = 3;
                                          // cv-qualified (initialized as required)
                                          // ill-formed: attempt to modify const
     ci = 4;
     int i = 2;
                                          // not cv-qualified
     const int* cip;
                                          // pointer to const int
                                          // OK: cv-qualified access path to unqualified
     cip = &i;
     *cip = 4;
                                          // ill-formed: attempt to modify through ptr to const
     int* ip;
                                          // cast needed to convert const int* to int*
     ip = const_cast<int*>(cip);
                                          // defined: *ip points to i, a non-const object
     const int* ciq = new const int (3);
                                                   // initialized as required
                                                   // cast required
     int* iq = const_cast<int*>(ciq);
     *iq = 4;
                                                   // undefined: modifies a const object
<sup>5</sup> For another example
     struct X {
       mutable int i;
       int j;
     };
     struct Y {
       X x;
       Y();
     };
     const Y y;
                                          // well-formed: mutable member can be modified
     y.x.i++;
```

<sup>92)</sup> 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.

<sup>6</sup> 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.

<sup>7</sup> [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]

The simple type specifiers are

```
simple-type-specifier:
       nested-name-specifier_{opt} type-name
       nested\text{-}name\text{-}specifier \ \mathtt{template} \ simple\text{-}template\text{-}id
       char
       char16_t
       char32_t
       wchar_t
       bool
       short
       int
       long
       signed
       unsigned
       float
       double
       void
       auto
       decltype-specifier
type-name:
       class\text{-}name
       enum-name
       typedef-name
       simple\mbox{-}template\mbox{-}id
decltype-specifier:
       decltype ( 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 10 summarizes the valid combinations of *simple-type-specifiers* and the types they specify.
- 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]

 $\odot$  ISO/IEC N3337

Table 10 — simple-type-specifiers and the types they specify

Specifier(s)	Type
type-name	the type named
simple-template-id	the type as defined in 14.2
char	"char"
unsigned char	"unsigned char"
signed char	"signed char"
char16_t	"char16_t"
char32_t	"char32_t"
bool	"bool"
unsigned	"unsigned int"
unsigned int	"unsigned int"
signed	"int"
signed int	"int"
int	"int"
unsigned short int	"unsigned short int"
unsigned short	"unsigned short int"
unsigned long int	"unsigned long int"
unsigned long	"unsigned long int"
unsigned long long int	"unsigned long long int"
unsigned long long	"unsigned long long int"
signed long int	"long int"
signed long	"long int"
signed long long int	"long long int"
signed long long	"long long int"
long long int	"long long int"
long long	"long long int"
long int	"long int"
long	"long int"
signed short int	"short int"
signed short	"short int"
short int	"short int"
short	"short int"
wchar_t	"wchar_t"
float	"float"
double	"double"
long double	"long double"
void	"void"
auto	placeholder for a type to be deduced
decltype(expression)	the type as defined below

- The type denoted by decltype(e) is defined as follows:
  - if e is an unparenthesized *id-expression* or an unparenthesized class member access (5.2.5), decltype(e) is the type of the entity named by e. If there is no such entity, or if e names a set of overloaded functions, the program is ill-formed;
  - otherwise, if e is an xvalue, decltype(e) is T&&, where T is the type of e;
  - 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:

 $-end \ example$ 

[Note: in the case where the operand of a decltype-specifier is a function call and the return type of the function is a class type, a special rule (5.2.2) ensures that the return type is not required to be complete (as it would be if the call appeared in a sub-expression or outside of a decltype-specifier). In this context, the common purpose of writing the expression is merely to refer to its type. In that sense, a decltype-specifier is analogous to a use of a typedef-name, so the usual reasons for requiring a complete type do not apply. In particular, it is not necessary to allocate storage for a temporary object or to enforce the semantic constraints associated with invoking the type's destructor. [Example:

```
template<class T> struct A { ~A() = delete; };
template<class T> auto h()
  -> A<T>;
                                    // identity
template < class T > auto i(T)
  -> T;
template < class T > auto f(T)
  -> decltype(i(h<T>()));
                                    // forces completion of A<T> and implicitly uses
                                    // A<T>::~A() for the temporary introduced by the
                                    // use of h(). (A temporary is not introduced
                                    // as a result of the use of i().)
template < class T > auto f(T)
                                    // #2
  -> void;
auto g() -> void {
  f(42);
                                    // OK: calls #2. (#1 is not a viable candidate: type
                                    // deduction fails (14.8.2) because A<int>::~A()
                                    // is implicitly used in its decltype-specifier)
}
template < class T > auto q(T)
                                    // does not force completion of A<T>; A<T>::~A() is
  -> decltype((h<T>()));
                                    // not implicitly used within the context of this decltype-specifier
void r() {
                                    // Error: deduction against q succeeds, so overload resolution
  q(42);
                                    // selects the specialization "q(T) -> decltype((h<T>())) [with T=int]".
                                    // The return type is A<int>, so a temporary is introduced and its
```

```
\label{eq:continuous} $$//\ destructor\ is\ used,\ so\ the\ program\ is\ ill\mbox{-}formed.$$$\\ --\ end\ example\ ]\ --\ end\ note\ ]
```

#### 7.1.6.3 Elaborated type specifiers

[dcl.type.elab]

```
elaborated-type-specifier: class-key \ attribute-specifier-seq_{opt} \ nested-name-specifier_{opt} \ identifier \\ class-key \ nested-name-specifier_{opt} {\tt template}_{opt} \ simple-template-id \\ {\tt enum} \ nested-name-specifier_{opt} \ identifier
```

An attribute-specifier-seq shall not appear in an elaborated-type-specifier unless the latter is the sole constituent of a declaration. If an elaborated-type-specifier is the sole constituent of a declaration, the declaration is ill-formed unless it is an explicit specialization (14.7.3), an explicit instantiation (14.7.2) or it has one of the following forms:

```
class-key attribute-specifier-seq_{opt} identifier; friend class-key::_{opt} identifier; friend class-key::_{opt} simple-template-id; friend class-key nested-name-specifier identifier; friend class-key nested-name-specifier template_{opt} simple-template-id;
```

In the first case, the *attribute-specifier-seq*, if any, appertains to the class being declared; the attributes in the *attribute-specifier-seq* are thereafter considered attributes of the class whenever it is named.

3.4.4 describes how name lookup proceeds for the *identifier* in an *elaborated-type-specifier*. If the *identifier* resolves to a *class-name* or *enum-name*, the *elaborated-type-specifier* introduces it into the declaration the same way a *simple-type-specifier* introduces its *type-name*. If the *identifier* resolves to a *typedef-name* or the *simple-template-id* resolves to an alias template specialization, the *elaborated-type-specifier* is ill-formed. [Note: This implies that, within a class template with a template type-parameter T, the declaration

```
friend class T;
```

is ill-formed. However, the similar declaration friend T; is allowed (11.3). — end note

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

[dcl.spec.auto]

- 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.
- <sup>2</sup> 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.
- 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:

```
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]
```

<sup>4</sup> 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).

- <sup>5</sup> A program that uses auto in a context not explicitly allowed in this section is ill-formed.
- Once the type of a *declarator-id* has been determined according to 8.3, the type of the declared variable using the *declarator-id* is determined from the type of its initializer using the rules for template argument deduction. Let T be the type that has been determined for a variable identifier d. Obtain P from T by replacing the occurrences of auto with 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:

<sup>7</sup> If the list of declarators contains more than one declarator, the type of each declared variable is determined as described above. If the type deduced for the template parameter U is not the same in each deduction, the program is ill-formed.

```
[ Example: const auto &i = expr;
```

The type of i is the deduced type of the parameter u in the call f(expr) of the following invented function template:

```
template <class U> void f(const U& u);
--end example]
```

## 7.2 Enumeration declarations

[dcl.enum]

An enumeration is a distinct type (3.9.2) with named constants. Its name becomes an *enum-name*, within its scope.

```
enum-name:
    identifier

enum-specifier:
    enum-head { enumerator-list_opt} }
    enum-head { enumerator-list , } 
enum-head:
    enum-key attribute-specifier-seq_opt identifier_opt enum-base_opt enum-key attribute-specifier-seq_opt nested-name-specifier identifier enum-base_opt

opaque-enum-declaration:
    enum-key attribute-specifier-seq_opt identifier enum-base_opt;
enum-key:
    enum enum class
    enum struct
```

```
enum-base:
    : type-specifier-seq
enumerator-list:
    enumerator-definition
    enumerator-definition:
enumerator-definition:
enumerator
enumerator = constant-expression
enumerator:
    identifier
```

The optional attribute-specifier-seq in the enum-head and the opaque-enum-declaration appertains to the enumeration; the attributes in that attribute-specifier-seq are thereafter considered attributes of the enumeration whenever it is named.

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. 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]

- 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.
- <sup>4</sup> If the *enum-key* is followed by a *nested-name-specifier*, the *enum-specifier* shall refer to an enumeration that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers (i.e., neither inherited nor introduced by a *using-declaration*), and the *enum-specifier* shall appear in a namespace enclosing the previous declaration.
- Each enumeration defines a type that is different from all other types. Each enumeration also has an underlying type. The underlying type can be explicitly specified using *enum-base*; if not explicitly specified, the underlying type of a scoped enumeration type is int. In these cases, the underlying type is said to be *fixed*. Following the closing brace of an *enum-specifier*, each enumerator has the type of its enumeration. If the underlying type is fixed, the type of each *enumerator* prior to the closing brace is the underlying type and the *constant-expression* in the *enumerator-definition* shall be a converted constant expression of the underlying type (5.19); 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 and the *constant-expression* shall be an integral constant expression (5.19).

§ 7.2 150

— 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  $e_{min}$  is the smallest enumerator and  $e_{max}$  is the largest, the values of the enumeration are the values in the range  $b_{min}$  to  $b_{max}$ , defined as follows: Let K be 1 for a two's complement representation and 0 for a one's complement or sign-magnitude representation.  $b_{max}$  is the smallest value greater than or equal to  $max(|e_{min}| K, |e_{max}|)$  and equal to  $2^M 1$ , where M is a non-negative integer.  $b_{min}$  is zero if  $e_{min}$  is non-negative and  $-(b_{max} + K)$  otherwise. The size of the smallest bit-field large enough to hold all the values of the enumeration type is max(M, 1) if  $b_{min}$  is zero and M + 1 otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators. If the enumerator-list is empty, the values of the enumeration are as if the enumeration had a single enumerator with value 0.93
- <sup>8</sup> Two enumeration types are layout-compatible if they have the same underlying type.
- The value of an enumerator or an object of an unscoped enumeration type is converted to an integer by integral promotion (4.5). [Example:

```
enum color { red, yellow, green=20, blue };
color col = red;
color* cp = &col;
if (*cp == blue) // ...
```

makes color a type describing various colors, and then declares col as an object of that type, and cp as a pointer to an object of that type. The possible values of an object of type color are red, yellow, green, blue; these values can be converted to the integral values 0, 1, 20, and 21. Since enumerations are distinct types, objects of type color can be assigned only values of type color.

Note that this implicit enum to int conversion is not provided for a scoped enumeration:

§ 7.2 151

<sup>93)</sup> This set of values is used to define promotion and conversion semantics for the enumeration type. It does not preclude an expression of enumeration type from having a value that falls outside this range.

Each enum-name and each unscoped enumerator is declared in the scope that immediately contains the enum-specifier. Each scoped enumerator is declared in the scope of the enumeration. These names obey the scope rules defined for all names in (3.3) and (3.4). [Example:

```
enum direction { left='l', right='r' };
  void g() {
    direction d;
                                     // OK
    d = left;
                                     // OK
    d = direction::right;
                                     //OK
  enum class altitude { high='h', low='l' };
  void h() {
                                     // OK
    altitude a;
    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 \rightarrow (arrow)), see 5.2.5. [Example:
  struct X {
    enum direction { left='l', right='r' };
    int f(int i) { return i==left ? 0 : i==right ? 1 : 2; }
  };
  void g(X* p) {
    direction d;
                                     // error: direction not in scope
    int i;
    i = p \rightarrow f(left);
                                     // error: left not in scope
    i = p->f(X::right);
                                     // OK
    i = p \rightarrow f(p \rightarrow left);
                                     //OK
    // ...
— end example]
```

#### 7.3 Namespaces

[basic.namespace]

A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units.

<sup>2</sup> The outermost declarative region of a translation unit is a namespace; see 3.3.6.

## 7.3.1 Namespace definition

[namespace.def]

```
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
```

<sup>2</sup> 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*.

- <sup>3</sup> The original-namespace-name in an extension-namespace-definition shall have previously been defined in an original-namespace-definition in the same declarative region.
- <sup>4</sup> Every namespace-definition shall appear in the global scope or in a namespace scope (3.3.6).
- <sup>5</sup> Because a namespace-definition contains declarations in its namespace-body and a namespace-definition is itself a declaration, it follows that namespace-definitions can be nested. [Example:

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

 $-end \ example$ ]
The  $enclosing \ nat$ 

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]

- <sup>7</sup> If the optional initial inline keyword appears in a namespace-definition for a particular namespace, that namespace is declared to be an *inline namespace*. The inline keyword may be used on an *extension-namespace-definition* only if it was previously used on the *original-namespace-definition* for that namespace.
- 8 Members of an inline namespace can be used in most respects as though they were members of the enclosing namespace. Specifically, the inline namespace and its enclosing namespace are both added to the set of

associated namespaces used in argument-dependent lookup (3.4.2) whenever one of them is, and a *using-directive* (7.3.4) that names the inline namespace is implicitly inserted into the enclosing namespace as for an unnamed namespace (7.3.1.1). Furthermore, each member of the inline namespace can subsequently be explicitly instantiated (14.7.2) or explicitly specialized (14.7.3) as though it were a member of the enclosing namespace. Finally, looking up a name in the enclosing namespace via explicit qualification (3.4.3.2) will include members of the inline namespace brought in by the *using-directive* even if there are declarations of that name in the enclosing namespace.

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

#### 7.3.1.1 Unnamed namespaces

[namespace.unnamed]

An unnamed-namespace-definition behaves as if it were replaced by inline<sub>opt</sub>namespace unique { /\* empty body \*/ } using namespace unique ; namespace unique { namespace-body }

where inline appears if and only if it appears in the *unnamed-namespace-definition*, all occurrences of *unique* in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the entire program.<sup>94</sup> [Example:

```
namespace { int i; }
                                   // unique ::i
 void f() { i++; }
                                   // unique ::i++
 namespace A {
   namespace {
     int i;
                                   // A:: unique ::i
                                   // A:: unique ::j
     int j;
   }
   void g() { i++; }
                                   // A:: unique ::i++
 using namespace A;
 void h() {
                                   // error: unique ::i or A:: unique ::i
   i++;
   A::i++;
                                   // A:: unique ::i
   j++;
                                   // A:: unique ::j
— end example]
```

### 7.3.1.2 Namespace member definitions

[namespace.memdef]

Members (including explicit specializations of templates (14.7.3)) of a namespace can be defined within that namespace. [Example:

```
namespace X {
  void f() { /* ... */ }
}
```

— end example]

<sup>2</sup> 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:

§ 7.3.1.2

<sup>94)</sup> Although entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit.

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<sup>95</sup> the friend class or function is a member of the innermost enclosing namespace. The name of the friend is not found by unqualified lookup (3.4.1) or by qualified lookup (3.4.3) until a matching declaration is provided in that namespace scope (either before or after the class definition granting friendship). If a friend function is called, its name may be found by the name lookup that considers functions from namespaces and classes associated with the types of the function arguments (3.4.2). If the name in a friend declaration is neither qualified nor a template-id and the declaration is a function or an elaborated-type-specifier, the lookup to determine whether the entity has been previously declared shall not consider any scopes outside the innermost enclosing namespace. [Note: The other forms of friend declarations cannot declare a new member of the innermost enclosing namespace and thus follow the usual lookup rules. — end note] [Example:

```
// Assume f and g have not yet been defined.
void h(int);
template <class T> void f2(T);
namespace A {
  class X {
                                  // A::f(X) is a friend
    friend void f(X);
    class Y {
                                  // A::g is a friend
      friend void g();
      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::g
  void g() { f(x); }
                                  // definition of A::f
  void f(X) { /* ... */}
  void h(int) { /* ... */ }
                                  // definition of A::h
  // A::f, A::g and A::h are visible here and known to be friends
using A::x;
void h() {
```

95) this implies that the name of the class or function is unqualified.

§ 7.3.1.2

```
A::f(x);
                                    // error: f is not a member of A::X
   A::X::f(x);
   A::X::Y::g();
                                    // error: g is not a member of A::X::Y
— end example]
```

## 7.3.2 Namespace alias

[namespace.alias]

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

```
name space-alias:
       identifier
namespace-alias-definition:
      namespace identifier = qualified-namespace-specifier ;
qualified-namespace-specifier:
       nested\text{-}name\text{-}specifier_{opt}\ namespace\text{-}name
```

- 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
- In a declarative region, a namespace-alias-definition can be used to redefine a namespace-alias declared in that declarative region to refer only to the namespace to which it already refers. [Example: the following declarations are well-formed:

```
namespace Company_with_very_long_name { /* ... */ }
 namespace CWVLN = Company_with_very_long_name;
 namespace CWVLN = Company_with_very_long_name;
                                                          // OK: duplicate
 namespace CWVLN = CWVLN;
— end example]
```

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

[namespace.udecl]

A using-declaration introduces a name into the declarative region in which the using-declaration appears. using-declaration:

```
using typename opt nested-name-specifier unqualified-id;
using :: unqualified-id;
```

The member name specified in a using-declaration is declared in the declarative region in which the using-declaration appears. [Note: Only the specified name is so declared; specifying an enumeration name in a using-declaration does not declare its enumerators in the using-declaration's declarative region. — end note If a using-declaration names a constructor (3.4.3.1), it implicitly declares a set of constructors in the class in which the using-declaration appears (12.9); otherwise the name specified in a using-declaration is a synonym for the name of some entity declared elsewhere.

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

```
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]
```

<sup>3</sup> 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:

- end example]
- <sup>4</sup> [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.
- <sup>5</sup> A using-declaration shall not name a template-id. [Example:

- end example]
- <sup>6</sup> A using-declaration shall not name a namespace.
- <sup>7</sup> A using-declaration shall not name a scoped enumerator.
- 8 A using-declaration for a class member shall be a member-declaration. [Example:

```
— 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:

— 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
};
```

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);
}
```

— end example]

-end example

- [Note: Partial specializations of class templates are found by looking up the primary class template and then considering all partial specializations of that template. If a using-declaration names a class template, partial specializations introduced after the using-declaration are effectively visible because the primary template is visible (14.5.5). end note
- Since a using-declaration is a declaration, the restrictions on declarations of the same name in the same declarative region (3.3) also apply to using-declarations. [Example:

```
{\tt namespace} \ {\tt A} \ \{
  int x;
}
namespace B {
  int i;
  struct g { };
  struct x { };
  void f(int);
  void f(double);
                      // OK: hides struct g
  void g(char);
void func() {
  int i;
                      // error: i declared twice
  using B::i;
  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;
                       // OK: hides struct B::x
  using A::x;
                       // assigns to A::x
  x = 99:
  struct x x1;
                      // x1 has class type B::x
```

— end example]

14 If a function declaration in namespace scope or block scope has the same name and the same parameter types as a function introduced by a using-declaration, and the declarations do not declare the same function, the program is ill-formed. [Note: Two using-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() {
                      // B::f(int) and B::f(double)
   using B::f;
                      // C::f(int), C::f(double), and C::f(char)
   using C::f;
                      // calls C::f(char)
   f('h');
                      // error: ambiguous: B::f(int) or C::f(int)?
   f(1);
                      // error: f(int) conflicts with C::f(int) and B::f(int)
   void f(int);
 }
- end example ] - end note ]
```

When a using-declaration brings names from a base class into a derived class scope, member functions and member function templates in the derived class override and/or hide member functions and member function templates with the same name, parameter-type-list (8.3.5), cv-qualification, and ref-qualifier (if any) in a base class (rather than conflicting). [Note: For using-declarations that name a constructor, see 12.9. — end note] [Example:

```
struct B {
   virtual void f(int);
   virtual void f(char);
   void g(int);
   void h(int);
 };
 struct D : B {
   using B::f;
   void f(int);
                      // OK: D::f(int) overrides B::f(int);
   using B::g;
   void g(char);
                      // OK
   using B::h;
   void h(int);
                      // OK: D::h(int) hides B::h(int)
 };
 void k(D* p)
                      // calls D::f(int)
   p->f(1);
   p->f('a');
                      // calls B::f(char)
   p->g(1);
                      // calls B::g(int)
   p->g('a');
                      // calls D::g(char)
— end example]
```

<sup>16</sup> 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,

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

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

— end note
```

<sup>18</sup> 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:

<sup>19</sup> If a *using-declaration* uses the keyword typename and specifies a dependent name (14.6.2), the name introduced by the *using-declaration* is treated as a *typedef-name* (7.1.3).

## 7.3.4 Using directive

[namespace.udir]

 $using\mbox{-}directive:$ 

attribute-specifier-seqoptusing namespace nested-name-specifieropt namespace-name;

A using-directive shall not appear in class scope, but may appear in namespace scope or in block scope. [Note: When looking up a namespace-name in a using-directive, only namespace names are considered, see 3.4.6. — end note] The optional attribute-specifier-seq appertains to the using-directive.

- <sup>2</sup> 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]
- <sup>3</sup> A using-directive does not add any members to the declarative region in which it appears. [Example:

```
namespace A {
  int i;
  namespace B {
    namespace C {
      int i;
    using namespace A::B::C;
    void f1() {
                       // OK, C::i visible in B and hides A::i
      i = 5;
  }
  {\tt namespace}\ {\tt D}\ \{
    using namespace B;
    using namespace C;
    void f2() {
      i = 5;
                       // ambiguous, B::C::i or A::i?
  }
  void f3() {
    i = 5;
                       // uses A::i
  }
}
void f4() {
  i = 5;
                      // ill-formed; neither i is visible
```

— end example]

For unqualified lookup (3.4.1), the *using-directive* is transitive: if a scope contains a *using-directive* that nominates a second namespace that itself contains *using-directives*, the effect is as if the *using-directives* from the second namespace also appeared in the first. [*Note:* For qualified lookup, see 3.4.3.2. — end note] [Example:

```
For another example,
namespace A {
  int i;
}
namespace B {
  int i;
  int j;
  {\tt namespace}\ {\tt 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
                      // ambiguous: C::k or D::k
    int l = k;
    int m = i;
                      // B::i hides A::i
    int n = j;
                      // D::j hides B::j
  }
}
```

- end example]
- <sup>5</sup> 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;
 void f() {
                      // error: name X found in two namespaces
   X(1);
                      // okay: name g refers to the same entity
   g();
                      // okay: overload resolution selects A::h
   h();
 }
-end note
```

During overload resolution, all functions from the transitive search are considered for argument matching. The set of declarations found by the transitive search is unordered. [Note: In particular, the order in which namespaces were considered and the relationships among the namespaces implied by the using-directives do not cause preference to be given to any of the declarations found by the search. —end note] An ambiguity

exists if the best match finds two functions with the same signature, even if one is in a namespace reachable through using-directives in the namespace of the other.  $^{96}$  [ Example:

```
namespace D {
   int d1;
   void f(char);
 }
 using namespace D;
                       // OK: no conflict with D::d1
 int d1;
 namespace E {
   int e:
   void f(int);
 namespace D {
                       // namespace extension
   int d2;
   using namespace E;
   void f(int);
 void f() {
                       // error: ambiguous ::d1 or D::d1?
   d1++;
                       // OK
   ::d1++;
   D::d1++:
                       // OK
                       // OK: D::d2
   d2++;
                       // OK: E::e
   e++:
                      // error: ambiguous: D::f(int) or E::f(int)?
   f(1);
   f('a');
                       // OK: D::f(char)
— end example]
```

#### 7.4 The asm declaration

[dcl.asm]

An asm declaration has the form

asm-definition:

asm ( string-literal ) ;

The asm declaration is conditionally-supported; its meaning is implementation-defined. [Note: Typically it is used to pass information through the implementation to an assembler.  $-end\ note$ ]

#### 7.5 Linkage specifications

[dcl.link]

- All function types, function names with external linkage, and variable names with external linkage have a language linkage. [Note: Some of the properties associated with an entity with language linkage are specific to each implementation and are not described here. For example, a particular language linkage may be associated with a particular form of representing names of objects and functions with external linkage, or with a particular calling convention, etc. end note] The default language linkage of all function types, function names, and variable names is C++ language linkage. Two function types with different language linkages are distinct types even if they are otherwise identical.
- Linkage (3.5) between C++ and non-C++ code fragments can be achieved using a linkage-specification:

 $linkage ext{-}specification:$ 

extern string-literal {  $declaration-seq_{opt}$ } extern string-literal declaration

<sup>96)</sup> 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 string-literal indicates the required language linkage. This International Standard specifies the semantics for the string-literals "C" and "C++". Use of a string-literal other than "C" or "C++" is conditionally-supported, with implementation-defined semantics. [Note: Therefore, a linkage-specification with a string-literal that is unknown to the implementation requires a diagnostic. — end note] [Note: It is recommended that the spelling of the string-literal be taken from the document defining that language. For example, Ada (not ADA) and Fortran or FORTRAN, depending on the vintage. — end note]

<sup>3</sup> Every implementation shall provide for linkage to functions written in the C programming language, "C", and linkage to C++ functions, "C++". [Example:

— end example

Linkage specifications nest. When linkage specifications nest, the innermost one determines the language linkage. A linkage specification does not establish a scope. A linkage-specification shall occur only in namespace scope (3.3). In a linkage-specification, the specified language linkage applies to the function types of all function declarators, function names with external linkage, and variable names with external linkage declared within the linkage-specification. [Example:

```
extern "C" void f1(void(*pf)(int));
                                    // the name f1 and its function type have C language
                                    // linkage; pf is a pointer to a C function
extern "C" typedef void FUNC();
FUNC f2;
                                    // the name f2 has C++ language linkage and the
                                    // function's type has C language linkage
extern "C" FUNC f3;
                                    // the name of function f3 and the function's type
                                    // have C language linkage
                                    // the name of the variable pf2 has C++ linkage and
void (*pf2)(FUNC*);
                                    // the type of pf2 is pointer to C++ function that
                                    // takes one parameter of type pointer to C function
extern "C" {
                                    // the name of the function f4 has
  static void f4();
                                    // internal linkage (not C language
                                    // linkage) and the function's type
                                    // 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.
}
                                    // OK: Name linkage (internal)
extern void f4();
                                    // 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 in determining the language linkage of the names of class members and the function type of class member functions. [Example:

```
extern "C" typedef void FUNC_c();
class C {
                                    // the name of the function mf1 and the member
   void mf1(FUNC_c*);
                                    // function's type have C++ language linkage; the
                                    // parameter has type pointer to C function
                                    // the name of the function mf2 and the member
   FUNC_c mf2;
                                    // 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 {
                                    // the name of the function mf and the member
  void mf();
                                    // function's type have C++ language linkage
  void mf2(void(*)());
                                    // the name of the function mf2 has C++ language
                                    // linkage; the parameter has type pointer to
                                    // C function
  };
}
```

- end example]
- If two declarations declare functions with the same name and parameter-type-list (8.3.5) to be members of the same namespace or declare objects with the same name to be members of the same namespace and the declarations give the names different language linkages, the program is ill-formed; no diagnostic is required if the declarations appear in different translation units. Except for functions with C++ linkage, a function declaration without a linkage specification shall not precede the first linkage specification for that function. A function can be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.
- At most one function with a particular name can have C language linkage. Two declarations for a function with C language linkage with the same function name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same function. Two declarations for a variable with C language linkage with the same name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same variable. An entity with C language linkage shall not be declared with the same name as an entity in global scope, unless both declarations denote the same entity; no diagnostic is required if the declarations appear in different translation units. A variable with C language linkage shall not be declared with the same name as a function with C language linkage (ignoring the namespace names that qualify the respective names); no diagnostic is required if the declarations appear in different translation units. [Note: Only one definition for an entity with a given name with C language linkage may appear in the program (see 3.2); this implies that such an entity must not be defined in more than one namespace scope. end note [Example:

```
int x;
namespace A {
  extern "C" int f();
  extern "C" int g() { return 1; }
  extern "C" int h();
  extern "C" int x();  // ill-formed: same name as global-space object x
```

<sup>7</sup> 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:

- end example]
- 8 [Note: Because the language linkage is part of a function type, when a pointer to C function (for example) is dereferenced, the function to which it refers is considered a C function. end note]
- <sup>9</sup> Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved.

7.6 Attributes [dcl.attr]

#### 7.6.1 Attribute syntax and semantics

[dcl.attr.grammar]

Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

```
attribute-specifier-seq:
       attribute-specifier-seq_{opt} attribute-specifier
attribute-specifier:
       [ [ attribute-list ] ]
       alignment-specifier
alignment-specifier:
       alignas ( type\text{-}id \dots _{opt} )
       alignas ( alignment-expression ... opt)
attribute-list:
       attribute_{opt}
       attribute-list, attribute_{opt}
       attribute \dots
       attribute-list, attribute...
attribute:
       attribute-token attribute-argument-clause_{opt}
attribute	ext{-}token:
       identifier
       attribute	ext{-}scoped	ext{-}token
```

```
attribute-scoped-token:
   attribute-namespace :: identifier
attribute-namespace:
   identifier
attribute-argument-clause:
   (balanced-token-seq)
balanced-token-seq:
   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 [Note: For each individual attribute, the form of the balanced-token-seq will be specified. —end note]

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]

- <sup>4</sup> Each attribute-specifier-seq is said to appertain to some entity or statement, identified by the syntactic context where it appears (Clause 6, Clause 7, Clause 8). If an attribute-specifier-seq that appertains to some entity or statement contains an attribute that is not allowed to apply to that entity or statement, the program is ill-formed. If an attribute-specifier-seq appertains to a friend declaration (11.3), that declaration shall be a definition. No attribute-specifier-seq shall appertain to an explicit instantiation (14.7.2).
- <sup>5</sup> 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:

#### 7.6.2 Alignment specifier

— end example]

[dcl.align]

An alignment-specifier may be applied to a variable or to a class data member, but it shall not be applied to a bit-field, a function parameter, the formal parameter of a catch clause (15.3), or a variable declared with the register storage class specifier. An alignment-specifier may also be applied to the declaration of a class or enumeration type. An alignment-specifier with an ellipsis is a pack expansion (14.5.3).

When the alignment-specifier is of the form alignas (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-specifier is of the form alignas (type-id), it shall have the same effect as alignas (alignof (type-id)) (5.3.6).
- <sup>4</sup> When multiple *alignment-specifiers* are specified for an entity, the alignment requirement shall be set to the strictest specified alignment.
- The combined effect of all *alignment-specifiers* in a declaration shall not specify an alignment that is less strict than the alignment that would be required for the entity being declared if all *alignment-specifiers* were omitted (including those in other declarations).
- If the defining declaration of an entity has an alignment-specifier, any non-defining declaration of that entity shall either specify equivalent alignment or have no alignment-specifier. Conversely, if any declaration of an entity has an alignment-specifier, every defining declaration of that entity shall specify an equivalent alignment. No diagnostic is required if declarations of an entity have different alignment-specifiers in different translation units.

[Example: An aligned buffer with an alignment requirement of A and holding N elements of type T other than char, signed char, or unsigned char can be declared as:

```
alignas(T) alignas(A) T buffer[N];
```

Specifying alignas(T) ensures that the final requested alignment will not be weaker than alignof(T), and therefore the program will not be ill-formed. — end example

8 [Example:

#### 7.6.3 Noreturn attribute

[dcl.attr.noreturn]

The attribute-token noreturn specifies that a function does not return. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute may be applied to the declarator-id in a function declaration. The first declaration of a function shall specify the noreturn

attribute if any declaration of that function specifies the **noreturn** attribute. If a function is declared with the **noreturn** attribute in one translation unit and the same function is declared without the **noreturn** attribute in another translation unit, the program is ill-formed; no diagnostic required.

- If a function f is called where f was previously declared with the noreturn attribute and f eventually returns, the behavior is undefined. [Note: The function may terminate by throwing an exception. end note] [Note: Implementations are encouraged to issue a warning if a function marked [[noreturn]] might return. end note]
- <sup>3</sup> [Example:

## — end example]

#### 7.6.4 Carries dependency attribute

[dcl.attr.depend]

- The attribute-token carries\_dependency specifies dependency propagation into and out of functions. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present. The attribute may be applied to the declarator-id of a parameter-declaration in a function declaration or lambda, in which case it specifies that the initialization of the parameter carries a dependency to (1.10) each lyalue-to-ryalue 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.
- The first declaration of a function shall specify the carries\_dependency attribute for its declarator-id if any declaration of the function specifies the carries\_dependency attribute. Furthermore, the first declaration of a function shall specify the carries\_dependency attribute for a parameter if any declaration of that function specifies the carries\_dependency attribute for that parameter. If a function or one of its parameters is declared with the carries\_dependency attribute in its first declaration in one translation unit and the same function or one of its parameters is declared without the carries\_dependency attribute in its first declaration in another translation unit, the program is ill-formed; no diagnostic required.
- Note: The carries\_dependency attribute does not change the meaning of the program, but may result in generation of more efficient code. end note
- 4 [Example:

```
/* Translation unit A. */
struct foo { int* a; int* b; };
std::atomic<struct foo *> foo_head[10];
int foo_array[10][10];

[[carries_dependency]] struct foo* f(int i) {
   return foo_head[i].load(memory_order_consume);
}

[[carries_dependency]] int g(int* x, int* y) {
   return kill_dependency(foo_array[*x][*y]);
}

/* Translation unit B. */
```

```
[[carries_dependency]] struct foo* f(int i);
[[carries_dependency]] int* g(int* x, int* y);
int c = 3;

void h(int i) {
   struct foo* p;

   p = f(i);
   do_something_with(g(&c, p->a));
   do_something_with(g(p->a, &c));
}
```

The carries\_dependency attribute on function f means that the return value carries a dependency out of f, so that the implementation need not constrain ordering upon return from f. Implementations of f and its caller may choose to preserve dependencies instead of emitting hardware memory ordering instructions (a.k.a. fences).

Function g's second 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

# [dcl.decl]

A declarator declares a single variable, function, or type, within a declaration. The *init-declarator-list* appearing in a declaration is a comma-separated sequence of declarators, each of which can have an initializer.

```
init-declarator-list: init-declarator init-declarator-list , init-declarator init-declarator: declarator init-declarator i
```

- 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.
- <sup>3</sup> Each *init-declarator* in a declaration is analyzed separately as if it was in a declaration by itself. <sup>97</sup>
- <sup>4</sup> Declarators have the syntax

```
declarator:
       ptr-declarator
       noptr-declarator\ parameters-and-qualifiers\ trailing-return-type
ptr-declarator:
       noptr-declarator
       ptr-operator ptr-declarator
noptr-declarator:
       declarator-id attribute-specifier-seq<sub>opt</sub>
       noptr-declarator parameters-and-qualifiers
       noptr-declarator [ constant-expression_{opt}] attribute-specifier-seq_{opt}
       ( ptr-declarator )
parameters-and-qualifiers:
       ( parameter-declaration-clause ) attribute-specifier-seq_{opt} cv-qualifier-seq_{opt}
              ref-qualifier_{opt} exception-specification_{opt}
trailing-return-type:
       -> trailing-type-specifier-seq abstract-declarator<sub>opt</sub>
```

97) A declaration with several declarators is usually equivalent to the corresponding sequence of declarations each with a single declarator. That is

```
T D1, D2, ... Dn;
is usually equivalent to
T D1; T D2; ... T Dn;
```

where T is a decl-specifier-seq and each Di is an init-declarator. An exception occurs when a name introduced by one of the declarators hides a type name used by the decl-specifiers, so that when the same decl-specifiers are used in a subsequent declaration, they do not have the same meaning, as in

```
struct S ...;
S S, T; // declare two instances of struct S
which is not equivalent to
    struct S ...;
S S;
S T; // error
Another exception occurs when T is auto (7.1.6.4), for example:
    auto i = 1, j = 2.0; // error: deduced types for i and j do not match
as opposed to
    auto i = 1; // OK: i deduced to have type int
    auto j = 2.0; // OK: j deduced to have type double
```

Declarators 172

```
ptr-operator:
       * attribute-specifier-seq_{opt} cv-qualifier-seq_{opt}
       & attribute-specifier-seq<sub>opt</sub>
       && attribute-specifier-seq_{opt}
       nested-name-specifier * attribute-specifier-seq_{opt} cv-qualifier-seq_{opt}
cv-qualifier-seq:
       cv-qualifier cv-qualifier-seq_{opt}
cv-qualifier:
       const
       volatile
ref-qualifier:
       Хr.
       &&
declarator-id:
       \dots_{opt} id-expression
       nested-name-specifier_{opt} 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).

The optional attribute-specifier-seq in a trailing-return-type appertains to the indicated return type. The type-id in a trailing-return-type includes the longest possible sequence of abstract-declarators. [Note: This resolves the ambiguous binding of array and function declarators. [Example:

#### 8.1 Type names

[dcl.name]

To specify type conversions explicitly, and as an argument of sizeof, alignof, new, or typeid, the name of a type shall be specified. This can be done with a *type-id*, which is syntactically a declaration for a variable or function of that type that omits the name of the entity.

```
type-id:
      type-specifier-seq abstract-declarator_{opt}
abstract-declarator:
      ptr-abstract-declarator
      noptr-abstract-declarator<sub>opt</sub> parameters-and-qualifiers trailing-return-type
      abstract-pack-declarator
ptr-abstract-declarator:
      noptr-abstract-declarator
      ptr-operator ptr-abstract-declarator_{opt}
noptr-abstract-declarator:
      noptr-abstract-declarator_{opt}\ parameters-and-qualifiers
      noptr-abstract-declarator_{opt} [ constant-expression_{opt} ] attribute-specifier-seq_{opt}
      ( ptr-abstract-declarator )
abstract-pack-declarator:
      noptr-abstract-pack-declarator
      ptr-operator abstract-pack-declarator
noptr-abstract-pack-declarator:
      noptr-abstract-pack-declarator\ parameters-and-qualifiers
      noptr-abstract-pack-declarator [ constant-expression_{opt} ] attribute-specifier-seq_{opt}
```

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:

§ 8.1 173

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

A type can also be named (often more easily) by using a typedef (7.1.3).

#### 8.2 Ambiguity resolution

[dcl.ambig.res]

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:

- end example]
- <sup>2</sup> 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:

```
#include <cstddef>
     char *p;
     void *operator new(std::size_t, int);
     void foo()
       const int x = 63;
                                         // new-placement expression
       new (int(*p)) int;
                                         // new type-id
       new (int(*[x]));
<sup>4</sup> For another example,
     template <class T>
     struct S {
       T *p;
     };
     S<int()> x;
                                         // type-id
                                         // expression (ill-formed)
     S<int(1)> y;
```

<sup>5</sup> For another example,

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

```
class C { };
                                    // void f(int(*fp)(C c)) { }
 void f(int(C)) { }
                                    // not: void f(int C);
 int g(C);
 void foo() {
                                    // error: cannot convert 1 to function pointer
   f(1);
                                    // OK
   f(g);
  For another example,
 class C { };
 void h(int *(C[10]));
                                    // void h(int *(*_fp)(C _parm[10]));
                                    // not: void h(int *C[10]);
— end example]
```

#### 8.3 Meaning of declarators

[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 declaratorid 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 (11.3). 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-seq following a declarator-id appertains to the entity that is declared.

A static, thread\_local, extern, register, mutable, friend, inline, virtual, or typedef specifier applies directly to each declarator-id in an init-declarator-list; the type specified for each declarator-id depends on both the decl-specifier-seq and its declarator.

Thus, a declaration of a particular identifier has the form

T D

where T is of the form attribute-specifier-seq 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.

<sup>4</sup> First, the decl-specifier-seq determines a type. In a declaration

ΤD

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]

- In a declaration attribute-specifier-seq<sub>opt</sub> 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

ci++;

\*pc = 2; cp = &ci;

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

8.3.1 Pointers [dcl.ptr]

- <sup>1</sup> In a declaration T D where D has the form
  - \* attribute-specifier-seq $_{opt}$  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-seq (7.6.1) appertains to the pointer and not to the object pointed to.

<sup>2</sup> [Example: the declarations

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

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

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

Examples of ill-formed operations are
ci = 1;  // error
```

// 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:

- $^3$  See also 5.17 and 8.5.
- <sup>4</sup> [Note: There are no pointers to references; see 8.3.2. Since the address of a bit-field (9.6) cannot be taken, a pointer can never point to a bit-field. end note]

8.3.2 References [dcl.ref]

In a declaration T D where D has either of the forms &  $attribute-specifier-seq_{opt} \mathtt{D1}$ 

&& attribute-specifier-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 reference to T." The optional attribute-specifier-seq appertains to the reference type. Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a typedef (7.1.3) or of a template type argument (14.3), in which case the cv-qualifiers are ignored. [Example:

```
typedef int& A;
const A aref = 3;  // ill-formed; 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.
- $^{3}$  [Example:

```
void f(double& a) { a += 3.14; }
// ...
double d = 0;
f(d);

declares a to be a reference parameter of f so the call f(d) will add 3.14 to d.
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```

declares the function g() to return a reference to an integer so g(3)=7 will assign 7 to the fourth element of the array v. For another example,

```
struct link {
   link* next;
};
link* first;
```

```
void h(link*& p) {  // p is a reference to pointer
  p->next = first;
  first = p;
  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).
- 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;
                                    // r1 has the type int&
 LRI\& r1 = i;
                                    // r2 has the type int&
 const LRI& r2 = i;
 const LRI&& r3 = i;
                                    // r3 has the type int&
 RRI& r4 = i;
                                    // r4 has the type int&
 RRI&& r5 = 5;
                                    // r5 has the type int&&
                                   // r6 has the type int&
 decltype(r2) \& r6 = i;
 decltype(r2)\&\& r7 = i;
                                   // r7 has the type int&
- end example]
```

#### 8.3.3 Pointers to members

[dcl.mptr]

In a declaration T D where D has the form

nested-name-specifier\* attribute-specifier- $seq_{opt}$  cv-qualifier- $seq_{opt}$  D1

and the nested-name-specifier denotes a class, and the type of the identifier in the declaration T D1 is "derived-declarator-type-list T", then the type of the identifier of D is "derived-declarator-type-list cv-qualifier-seq pointer to member of class nested-name-specifier of type T". The optional attribute-specifier-seq (7.6.1) appertains to the pointer-to-member.

<sup>2</sup> [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:

A pointer to member shall not point to a static member of a class (9.4), a member with reference type, or "cv void."

[Note: See also 5.3 and 5.5. The type "pointer to member" is distinct from the type "pointer", that is, a pointer to member is declared only by the pointer to member declarator syntax, and never by the pointer declarator syntax. There is no "reference-to-member" type in C++. — end note

8.3.4 Arrays [dcl.array]

In a declaration T D where D has the form

— end example]

D1 [ constant-expression $_{opt}$ ] attribute-specifier-seq $_{opt}$ 

and the type of the identifier in the declaration T D1 is "derived-declarator-type-list T", then the type of the identifier of D is an array type; if the type of the identifier of D contains the auto type-specifier, the program is ill-formed. T is called the array element type; this type shall not be a reference type, the (possibly cv-qualified) type void, a function type or an abstract class type. If the constant-expression (5.19) is present, it shall be an integral constant expression and its value shall be greater than zero. The constant expression specifies the bound of (number of elements in) the array. If the value of the constant expression is N, the array has N elements numbered 0 to N-1, and the type of the identifier of D is "derived-declarator-type-list array of N T". An object of array type contains a contiguously allocated non-empty set of N subobjects of type T. Except as noted below, if the constant expression is omitted, the type of the identifier of D is "derived-declarator-type-list array of unknown bound of T", an incomplete object type. The type "derived-declarator-type-list array of N T" is a different type from the type "derived-declarator-type-list array of unknown bound of T", see 3.9. Any type of the form "cv-qualifier-seq array of N T" is adjusted to "array of N cv-qualifier-seq T", and similarly for "array of unknown bound of T". The optional attribute-specifier-seq appertains to the array. [Example:

- end example] [Note: An "array of N cv-qualifier-seq T" has cv-qualified type; see 3.9.3. end note]
- An array can be constructed from one of the fundamental types (except void), from a pointer, from a pointer to member, from a class, from an enumeration type, or from another array.
- When several "array of" specifications are adjacent, a multidimensional array is created; only the first of the constant expressions that specify the bounds of the arrays may be omitted. In addition to declarations in which an incomplete object type is allowed, an array bound may be omitted in some cases in the declaration of a function parameter (8.3.5). An array bound may also be omitted when the declarator is followed by an

initializer (8.5). In this case the bound is calculated from the number of initial elements (say, N) supplied (8.5.1), and the type of the identifier of D is "array of N T." Furthermore, if there is a preceding declaration of the entity in the same scope in which the bound was specified, an omitted array bound is taken to be the same as in that earlier declaration, and similarly for the definition of a static data member of a class.

4 [Example:

```
float fa[17], *afp[17];
```

declares an array of float numbers and an array of pointers to float numbers. For another example, static int x3d[3][5][7];

declares a static three-dimensional array of integers, with rank  $3 \times 5 \times 7$ . In complete detail, x3d is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions x3d, x3d[i], x3d[i][j], x3d[i][j][k] can reasonably appear in an expression. Finally,

- end example]
- <sup>5</sup> [Note: conversions affecting expressions of array type are described in 4.2. Objects of array types cannot be modified, see 3.10. end note]
- [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.
- 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

```
int x[3][5];
```

Here x is a  $3 \times 5$  array of integers. When x appears in an expression, it is converted to a pointer to (the first of three) five-membered arrays of integers. In the expression x[i] which is equivalent to \*(x+i), x is first converted to a pointer as described; then x+i is converted to the type of x, which involves multiplying i by the length of the object to which the pointer points, namely five integer objects. The results are added and indirection applied to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer. — end example i — end note i

*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]

8.3.5Functions [dcl.fct]

In a declaration T D where D has the form

parameter-declaration-clause) cv-qualifier-seq<sub>opt</sub>

ref-qualifier ont exception-specification ont attribute-specifier-seq ont

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) cvqualifier-seq<sub>opt</sub> ref-qualifier<sub>opt</sub> returning T". The optional attribute-specifier-seq appertains to the function type.

In a declaration T D where D has the form D1 ( parameter-declaration-clause )  $cv\text{-}qualifier\text{-}seq_{opt}$ 

 $ref-qualifier_{opt}\ exception-specification_{opt}\ attribute-specifier-seq_{opt}\ trailing-return-type$ 

and the type of the contained declarator-id in the declaration T D1 is "derived-declarator-type-list T", T shall be the single type-specifier auto. The type of the declarator-id in D is "derived-declarator-type-list function of (parameter-declaration-clause) cv-qualifier-seq<sub>opt</sub> ref-qualifier<sub>opt</sub> returning trailing-return-type". The optional attribute-specifier-seq appertains to the function type.

A type of either form is a function type.<sup>98</sup>

```
parameter-declaration-clause:
      parameter-declaration-list_{opt}..._{opt}
      parameter-declaration-list , ...
parameter-declaration-list:
      parameter-declaration
      parameter-declaration-list, parameter-declaration
parameter-declaration:
      attribute-specifier-seq decl-specifier-seq declarator
      attribute-specifier-seq decl-specifier-seq declarator = initializer-clause
      attribute-specifier-seq abstract-declarator<sub>opt</sub>
      attribute-specifier-seq abstract-declarator ont = initializer-clause
```

The optional attribute-specifier-seq in a parameter-declaration appertains to the parameter.

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]

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 parametertype-list. The type of a function is determined using the following rules. The type of each parameter

<sup>98)</sup> As indicated by syntax, cv-qualifiers are a significant component in function return types.

(including function parameter packs) is determined from its own decl-specifier-seq and declarator. After determining the type of each parameter, any parameter of type "array of T" or "function returning T" is adjusted to be "pointer to T" or "pointer to function returning T," respectively. After producing the list of parameter types, any top-level cv-qualifiers modifying a parameter type are deleted when forming the function type. The resulting list of transformed parameter types and the presence or absence of the ellipsis or a function parameter pack is the function's parameter-type-list. [Note: This transformation does not affect the types of the parameters. For example, int(\*) (const int p, decltype(p)\*) and int(\*) (int, const int\*) are identical types. -end note]

- 6 A cv-qualifier-seq or 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,
  - the top-level function type of a function typedef declaration or alias-declaration,
  - the type-id in the default argument of a type-parameter (14.1), or
  - the type-id of a template-argument for a type-parameter (14.2).

The effect of a *cv-qualifier-seq* in a function declarator is not the same as adding cv-qualification on top of the function type. In the latter case, the cv-qualifiers are ignored. [*Note:* a function type that has a *cv-qualifier-seq* is not a cv-qualified type; there are no cv-qualified function types. — *end note*] [*Example:* 

— end example] 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 pointers to functions, references to functions, and pointers to member functions. — end note]

<sup>7</sup> [Example: the declaration

```
int fseek(FILE*, long, int);
```

declares a function taking three arguments of the specified types, and returning int (7.1.6). — end example

- If the type of a parameter includes a type of the form "pointer to array of unknown bound of T" or "reference to array of unknown bound of T," the program is ill-formed.<sup>99</sup> 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.
- <sup>9</sup> 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).
- A typedef of function type may be used to declare a function but shall not be used to define a function (8.4). [Example:

<sup>99)</sup> 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.

— 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:

- 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
- $^{12}$  [ Example: the declaration

```
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]
```

A declarator-id or abstract-declarator containing an ellipsis shall only be used in a parameter-declaration. Such a parameter-declaration is a parameter pack (14.5.3). When it is part of a parameter-declaration-clause,

the parameter pack is a function parameter pack (14.5.3). [Note: Otherwise, the parameter-declaration is part of a template-parameter-list and the parameter pack is a template parameter pack; see 14.1. — end note] A function parameter pack is a pack expansion (14.5.3). [Example:

```
template<typename... T> void f(T (* ...t)(int, int));
int add(int, int);
float subtract(int, int);
void g() {
  f(add, subtract);
}
```

There is a syntactic ambiguity when an ellipsis occurs at the end of a parameter-declaration-clause without a preceding comma. In this case, the ellipsis is parsed as part of the abstract-declarator if the type of the parameter names a template parameter pack that has not been expanded; otherwise, it is parsed as part of the parameter-declaration-clause. 100

### 8.3.6 Default arguments

[dcl.fct.default]

- If an *initializer-clause* is specified in a *parameter-declaration* this *initializer-clause* is used as a default argument. Default arguments will be used in calls where trailing arguments are missing.
- <sup>2</sup> [Example: the declaration

— end example]

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

declares a function that can be called with zero, one, or two arguments of type int. It can be called in any of these ways:

```
point(1,2); point(1); point();
```

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

- A default argument shall be specified only in the parameter-declaration-clause of a function declaration or in a template-parameter (14.1); in the latter case, the initializer-clause shall be an assignment-expression. A default argument shall not be specified for a parameter pack. If it is specified in a parameter-declaration-clause, it shall not occur within a declarator or abstract-declarator of a parameter-declaration.<sup>101</sup>
- 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:

<sup>100)</sup> One can explicitly disambiguate the parse either by introducing a comma (so the ellipsis will be parsed as part of the parameter-declaration-clause) or by introducing a name for the parameter (so the ellipsis will be parsed as part of the declarator-id).

<sup>101)</sup> This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or typedef declarations.

```
}
void m() {
                                   // has no defaults
  void f(int, int);
                                   // error: wrong number of arguments
  f(4);
                                   // OK
  void f(int, int = 5);
                                   // OK, calls f(4, 5);
  f(4);
  void f(int, int = 5);
                                   // error: cannot redefine, even to
                                   // same value
}
void n() {
                                   // OK, calls f(6, 7)
  f(6);
}
```

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

A default argument is implicitly converted (Clause 4) to the parameter type. The default argument has the same semantic constraints as the initializer in a declaration of a variable of the parameter type, using the copy-initialization semantics (8.5). The names in the default argument are bound, and the semantic constraints are checked, at the point where the default argument appears. Name lookup and checking of semantic constraints for default arguments in function templates and in member functions of class templates are performed as described in 14.7.1. [Example: in the following code, g will be called with the value f(2):

 $-end\ example$ ] [Note: In member function declarations, names in default arguments are looked up as described in 3.4.1. Access checking applies to names in default arguments as described in Clause 11. -end note]

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

<sup>7</sup> Local variables shall not be used in a default argument. [Example:

```
void f() {
  int i;
  extern void g(int x = i);  //error
  // ...
}
```

— end example]

8 The keyword this shall not be used in a default argument of a member function. [Example:

```
class A {
   void f(A* p = this) { } // error
};
```

— end example]

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 a default argument, even if they are not evaluated. Parameters of a function declared before a default argument are in scope and can hide namespace and class member names. [Example:

— end example] Similarly, a non-static member shall not be used in a default argument, even if it is not evaluated, unless it appears as the id-expression of a class member access expression (5.2.5) or unless it is used to form a pointer to member (5.3.1). [Example: the declaration of X::mem1() in the following example is ill-formed because no object is supplied for the non-static member X::a used as an initializer.

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

```
int f(int = 0);

void h() {
   int j = f(1);
   int k = f();
}

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.

O 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:

#### 8.4 Function definitions

[dcl.fct.def]

#### 8.4.1 In general

[dcl.fct.def.general]

Function definitions have the form

function-definition:

 $attribute-specifier-seq_{opt} \ decl-specifier-seq_{opt} \ declarator \ virt-specifier-seq_{opt} \ function-body \\ function-body: \\ ctor-initializer_{opt} \ compound-statement \\ function-try-block$ 

= default; = delete;

Any informal reference to the body of a function should be interpreted as a reference to the non-terminal function-body. The optional attribute-specifier-seq in a function-definition appertains to the function. A virt-specifier-seq can be part of a function-definition only if it is a member-declaration (9.2).

The declarator in a function-definition shall have the form

D1 ( parameter-declaration-clause ) cv-qualifier-seq<sub>opt</sub>

ref-qualifier<sub>opt</sub> exception-specification<sub>opt</sub> attribute-specifier-seq<sub>opt</sub> trailing-return-type<sub>opt</sub> as described in 8.3.5. A function shall be defined only in namespace or class scope.

<sup>3</sup> [Example: a simple example of a complete function definition is

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

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

- <sup>4</sup> A ctor-initializer is used only in a constructor; see 12.1 and 12.6.
- <sup>5</sup> 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.
- 6 [Note: Unused parameters need not be named. For example,

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

§ 8.4.1

```
— end note]
```

<sup>7</sup> 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

```
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.  $^{102}$ 

[Example:

### 8.4.2 Explicitly-defaulted functions

[dcl.fct.def.default]

A function definition of the form:

attribute-specifier-seq $_{opt}$  decl-specifier-seq $_{opt}$  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, and
- not have default arguments.
- An explicitly-defaulted function may be declared **constexpr** only if it would have been implicitly declared as **constexpr**, and may have an explicit *exception-specification* only if it is compatible (15.4) with the *exception-specification* on the implicit declaration. If a function is explicitly defaulted on its first declaration,
  - it is implicitly considered to be constexpr if the implicit declaration would be,
  - it is implicitly considered to have the same *exception-specification* as if it had been implicitly declared (15.4), and
  - 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 [Example:

```
struct S {
  constexpr S() = default;
                                                // ill-formed: implicit S() is not constexpr
                                                // ill-formed: default argument
  S(int a = 0) = default;
  void operator=(const S&) = default;
                                                // ill-formed: non-matching return type
                                                // ill-formed: exception specification does not match
  ~S() throw(int) = default;
private:
  int i;
                                                // OK: private copy constructor
  S(S&);
};
                                                // OK: defines copy constructor
S::S(S\&) = default;
```

102) Implementations are permitted to provide additional predefined variables with names that are reserved to the implementation (17.6.4.3.2). If a predefined variable is not odr-used (3.2), its string value need not be present in the program image.

§ 8.4.2

```
— end example]
```

<sup>4</sup> 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 or deleted on its first declaration. A user-provided explicitly-defaulted function (i.e., explicitly defaulted after its first declaration) is defined at the point where it is explicitly defaulted; if such a function is implicitly defined as deleted, the program is ill-formed. [Note: Declaring a function as defaulted after its first declaration can provide efficient execution and concise definition while enabling a stable binary interface to an evolving code base. — end note]

<sup>5</sup> [Example:

```
struct trivial {
   trivial() = default;
   trivial(const trivial&) = default;
   trivial(trivial&&) = default;
   trivial& operator=(const trivial&) = default;
   trivial& operator=(trivial&&) = default;
   ~trivial() = default;
};

struct nontrivial1 {
   nontrivial1();
};
nontrivial1::nontrivial1() = default;  // not first declaration

— end example]
```

#### 8.4.3 Deleted definitions

[dcl.fct.def.delete]

<sup>1</sup> A function definition of the form:

— end example]

attribute-specifier-seq $_{opt}$  decl-specifier-seq $_{opt}$  declarator = 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]
- 3 [Example: One can enforce non-default initialization and non-integral initialization with

```
struct onlydouble {
                                         // OK, but redundant
    onlydouble() = delete;
    onlydouble(std::intmax_t) = delete;
    onlydouble(double);
  };
    — end example]
   Example: One can prevent use of a class in certain new expressions by using deleted definitions of a
user-declared operator new for that class.
  struct sometype {
    void *operator new(std::size_t) = delete;
    void *operator new[](std::size_t) = delete;
  };
                                   // error, deleted class operator new
  sometype *p = new sometype;
  sometype *q = new sometype[3]; // error, deleted class operator new[]
```

§ 8.4.3

[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.

```
struct moveonly {
  moveonly() = default;
  moveonly(const moveonly&) = delete;
  moveonly(moveonly&&) = default;
  moveonly& operator=(const moveonly&) = delete;
  moveonly& operator=(moveonly&&) = default;
  ~moveonly() = default;
};
moveonly *p;
moveonly *p;
moveonly q(*p); // error, deleted copy constructor

— end example]
```

<sup>4</sup> 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:

8.5 Initializers [dcl.init]

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).

```
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 }
    { }
}
```

Except for objects declared with the constexpr specifier, for which see 7.1.5, an *initializer* in the definition of a variable can consist of arbitrary expressions involving literals and previously declared variables and functions, regardless of the variable's storage duration. [Example:

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

— end example]
```

- <sup>3</sup> [Note: Default arguments are more restricted; see 8.3.6.
- The order of initialization of variables with static storage duration is described in 3.6 and 6.7. end note
- <sup>5</sup> 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 O (zero), taken as an integral constant expression, converted to T;<sup>103</sup>
  - if T is a (possibly cv-qualified) non-union class type, each non-static data member and each base-class subobject is zero-initialized and padding is initialized to zero bits;
  - if T is a (possibly cv-qualified) union type, the object's first non-static named data member is zero-initialized and padding is initialized to zero bits;
  - if T is an array type, each element is zero-initialized;
  - if T is a reference type, no initialization is performed.
- <sup>6</sup> 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.

- <sup>7</sup> 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.

An object that is value-initialized is deemed to be constructed and thus subject to provisions of this International Standard applying to "constructed" objects, objects "for which the constructor has completed," etc., even if no constructor is invoked for the object's initialization.

- 8 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,

X a():

<sup>103</sup>) As specified in 4.10, converting an integral constant expression whose value is 0 to a pointer type results in a null pointer value.

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

- <sup>11</sup> 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]
- 12 An initializer for a static member is in the scope of the member's class. [Example:

```
int a;
struct X {
    static int a;
    static int b;
};
int X::a = 1;
int X::b = a;
    // X::b = X::a

— end example]
```

- The form of initialization (using parentheses or =) is generally insignificant, but does matter when the initializer or the entity being initialized has a class type; see below. If the entity being initialized does not have class type, the *expression-list* in a parenthesized initializer shall be a single expression.
- 14 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 *copy-initialization*. [Note: Copy-initialization may invoke a move (12.8). — end note]

The initialization that occurs in the forms

```
T x(a);
T x{a};
```

as well as in **new** expressions (5.3.4), **static\_cast** expressions (5.2.9), functional notation type conversions (5.2.3), and base and member initializers (12.6.2) is called *direct-initialization*.

The semantics of initializers are as follows. The *destination type* is the type of the object or reference being initialized and the *source type* is the type of the initializer expression. If the initializer is not a single (possibly parenthesized) expression, the source type is not defined.

- If the initializer is a (non-parenthesized) braced-init-list, the object or reference 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 char16\_t, an array of char32\_t, or an array of 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 cv-qualified) class type:
  - If the initialization is direct-initialization, or if it is copy-initialization where the cv-unqualified version of the source type is the same class as, or a derived class of, the class of the destination, constructors are considered. The applicable constructors are enumerated (13.3.1.3), and the best one is chosen through overload resolution (13.3). The constructor so selected is called to initialize

- the object, with the initializer expression or *expression-list* as its argument(s). If no constructor applies, or the overload resolution is ambiguous, the initialization is ill-formed.
- Otherwise (i.e., for the remaining copy-initialization cases), user-defined conversion sequences that can convert from the source type to the destination type or (when a conversion function is used) to a derived class thereof are enumerated as described in 13.3.1.4, and the best one is chosen through overload resolution (13.3). If the conversion cannot be done or is ambiguous, the initialization is ill-formed. The function selected is called with the initializer expression as its argument; if the function is a constructor, the call initializes a temporary of the cv-unqualified version of the destination type. The temporary is a prvalue. The result of the call (which is the temporary for the constructor case) is then used to direct-initialize, according to the rules above, the object that is the destination of the copy-initialization. In certain cases, an implementation is permitted to eliminate the copying inherent in this direct-initialization by constructing the intermediate result directly into the object being initialized; see 12.2, 12.8.
- Otherwise, if the source type is a (possibly cv-qualified) class type, conversion functions are considered. The applicable conversion functions are enumerated (13.3.1.5), and the best one is chosen through overload resolution (13.3). The user-defined conversion so selected is called to convert the initializer expression into the object being initialized. If the conversion cannot be done or is ambiguous, the initialization is ill-formed.
- Otherwise, the initial value of the object being initialized is the (possibly converted) value of the initializer expression. Standard conversions (Clause 4) will be used, if necessary, to convert the initializer expression to the cv-unqualified version of the destination type; no user-defined conversions are considered. If the conversion cannot be done, the initialization is ill-formed. [Note: An expression of type "cv1 T" can initialize an object of type "cv2 T" independently of the cv-qualifiers cv1 and cv2.

```
int a;
const int b = a;
int c = b;

— end note]
```

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

#### 8.5.1 Aggregates

[dcl.init.aggr]

- 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).
- 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:

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

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

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

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

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

declares and initializes x as a one-dimensional array that has three elements since no size was specified and there are three initializers. — end example] An empty initializer list  $\{\}$  shall not be used as the initializer-clause for an array of unknown bound.  $^{104}$ 

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

```
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:

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

is ill-formed. — end example ]

If there are fewer *initializer-clauses* in the list than there are members in the aggregate, then each member not explicitly initialized shall be initialized from an empty initializer list (8.5.4). [Example:

```
struct S { int a; const char* b; int c; };
S ss = { 1, "asdf" };
```

initializes ss.a with 1, ss.b with "asdf", and ss.c with the value of an expression of the form int(), that is, 0. —end example]

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

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

104) The syntax provides for empty initializer-lists, but nonetheless C++ does not have zero length arrays.

```
— end example]
```

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

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:

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.<sup>105</sup> 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:

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
}:
```

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

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

The initializer for y begins with a left brace, but the one for y[0] does not, therefore three elements from the list are used. Likewise the next three are taken successively for y[1] and y[2]. — end example]

All implicit type conversions (Clause 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:

```
struct A {
  int i;
  operator int();
```

<sup>105)</sup> Braces cannot be elided in other uses of list-initialization.

```
};
struct B {
    A a1, a2;
    int z;
};
A a;
B b = { 4, a, a };
```

Braces are elided around the *initializer-clause* for b.a1.i. b.a1.i is initialized with 4, b.a2 is initialized with a, b.z is initialized with whatever a.operator int() returns. — end example

- [Note: An aggregate array or an aggregate class may contain members of a class type with a user-provided constructor (12.1). Initialization of these aggregate objects is described in 12.6.1. end note
- [Note: Whether the initialization of aggregates with static storage duration is static or dynamic is specified in 3.6.2 and 6.7. end note]
- 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:

— end example]

<sup>16</sup> [Note: As described above, the braces around the *initializer-clause* for a union member can be omitted if the union is a member of another aggregate. — end note]

#### 8.5.2 Character arrays

[dcl.init.string]

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\n";
```

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

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

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

is ill-formed since there is no space for the implied trailing '\0'. — end example

<sup>3</sup> 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]

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:

— end example]

<sup>2</sup> 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.

<sup>3</sup> 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:

```
int& r1;  // error: initializer missing extern int& r2;  // OK
```

— end example]

- 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
- <sup>5</sup> 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 Ivalue (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 Ivalue of type "cv3 T3," where "cv1 T1" is reference-compatible with "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 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]

— end example]

[Example:

106) This requires a conversion function (12.3.2) returning a reference type.

— Otherwise, the reference shall be an Ivalue reference to a non-volatile const type (i.e., cv1 shall be const), or the reference shall be an rvalue reference. [Example:

- If the initializer expression
  - is an xvalue, class prvalue, array prvalue or function lvalue and "cv1 T1" is reference-compatible with "cv2 T2", or
  - has a class type (i.e., T2 is a class type), where T1 is not reference-related to T2, and can be implicitly converted to an xvalue, class prvalue, or function lvalue of type "cv3 T3", where "cv1 T1" is reference-compatible with "cv3 T3",

then the reference is bound to the value of the initializer expression in the first case and to the result of the conversion in the second case (or, in either case, to an appropriate base class subobject). In the second case, if the reference is an rvalue reference and the second standard conversion sequence of the user-defined conversion sequence includes an lvalue-to-rvalue conversion, the program is ill-formed.

[ Example:

```
struct A { };
 struct B : A { } b;
 extern B f();
 const A\& rca2 = f();
                                          // bound to the A subobject of the B rvalue.
 A\&\& rra = f();
                                          // same as above
 struct X {
   operator B();
   operator int&();
 } x;
                                          // bound to the A subobject of the result of the conversion
 const A\& r = x;
 int i2 = 42;
 int&& rri = static_cast<int&&>(i2); // bound directly to i2
                                         // bound directly to the result of operator B
 B&& rrb = x;
 int\&\& rri2 = X();
                                         // error: lvalue-to-rvalue conversion applied to the
                                         // result of operator int&
— end example]
```

— Otherwise, a temporary of type "cv1 T1" is created and initialized from the initializer expression using the rules for a non-reference copy-initialization (8.5). The reference is then bound to the temporary. If T1 is reference-related to T2, cv1 shall be the same cv-qualification as, or greater cv-qualification than, cv2. If T1 is reference-related to T2 and the reference is an rvalue reference,

the initializer expression shall not be an lyalue. [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.

<sup>6</sup> [Note: 12.2 describes the lifetime of temporaries bound to references. — end note]

#### 8.5.4 List-initialization

[dcl.init.list]

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)
 — in a mem-initializer (12.6.2)
 — on the right-hand side of an assignment (5.17)
  [Example:
 int a = \{1\};
 std::complex<double> z{1,2};
 new std::vector<std::string>{"once", "upon", "a", "time"}; // 4 string elements
 f( {"Nicholas", "Annemarie"} ); // pass list of two elements
                                    // return list of one element
 return { "Norah" };
 int* e {};
                                    //\ initialization\ to\ zero\ /\ null\ pointer
                                    // explicitly construct a double
 x = double{1};
 std::map<std::string,int> anim = { {"bear",4}, {"cassowary",2}, {"tiger",7} };
- end example ] - end note ]
```

- 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.
- <sup>3</sup> 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 T is an aggregate, aggregate initialization is performed (8.5.1). [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. The applicable constructors are enumerated and the best one is chosen through overload resolution (13.3, 13.3.1.7). If a narrowing conversion (see below) is required to convert any of the arguments, the program is ill-formed.

[Example:

```
struct S {
    S(std::initializer_list<double>); // #1
    S(std::initializer_list<int>);
                                       // #3
    S();
    // ...
                                       // invoke #1
  S s1 = \{ 1.0, 2.0, 3.0 \};
                                       // invoke #2
  S s2 = { 1, 2, 3 };
 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
                                       // #1
    S(int, double, double);
    S();
                                       // #2
    // ...
  };
                                       // OK: invoke #1
  S s1 = { 1, 2, 3.0 };
  S s2 { 1.0, 2, 3 };
                                       // error: narrowing
  S s3 { };
                                       // OK: invoke #2
— end example]
```

— Otherwise, if T is a reference type, 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&);
 };
                                       // OK: invoke #1
 const S\& r1 = \{ 1, 2, 3.0 \};
                                       // OK: invoke #2
 const S& r2 { "Spinach" };
                                       // error: initializer is not an lvalue
 S\& r3 = \{ 1, 2, 3 \};
                                       //OK
 const int& i1 = { 1 };
                                       // error: narrowing
 const int& i2 = \{ 1.1 \};
                                       // OK: iar is bound to temporary array
 const int (&iar)[2] = { 1, 2 };
— end example]
```

— Otherwise, if the initializer list has a single element, the object or reference 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:
```

— 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; };
                                        // aggregate initialization
A a1 { 1, 2 };
A a2 { 1.2 };
                                        // error: narrowing
struct B {
 B(std::initializer_list<int>);
};
                                        // creates initializer_list<int> and calls constructor
B b1 { 1, 2 };
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
                                        // initialize to 1
int j { 1 };
int k { };
                                        // initialize to 0
```

```
— end example]
```

— end example]

Within the *initializer-list* of a *braced-init-list*, the *initializer-clauses*, including any that result from pack expansions (14.5.3), are evaluated in the order in which they appear. That is, every value computation and side effect associated with a given *initializer-clause* is sequenced before every value computation and side effect associated with any *initializer-clause* that follows it in the comma-separated list of the *initializer-list*. [Note: This evaluation ordering holds regardless of the semantics of the initialization; for example, it applies when the elements of the *initializer-list* are interpreted as arguments of a constructor call, even though ordinarily there are no sequencing constraints on the arguments of a call. — end note]

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.
```

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 ]

- <sup>7</sup> 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;
                              // OK, though it might narrow (in this case, it does narrow)
 char c1 = x;
                              // error: might narrow
 char c2\{x\};
                              //\ error:\ narrows\ (assuming\ {\tt char}\ is\ 8\ bits)
 char c3{y};
                              // OK: no narrowing needed
 char c4\{z\};
 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
                              // error: might narrow
 float f1 { x };
 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]
```

§ 8.5.4 203

9 Classes [class]

A class is a type. Its name becomes a class-name (9.1) within its scope.

```
class-name: identifier simple-template-id
```

Class-specifiers and elaborated-type-specifiers (7.1.6.3) are used to make class-names. An object of a class consists of a (possibly empty) sequence of members and base class objects.

```
class-specifier:
    class-head { member-specificationopt} }
class-head:
    class-key attribute-specifier-seqopt class-head-name class-virt-specifieropt base-clauseopt
    class-key attribute-specifier-seqopt base-clauseopt
class-head-name:
    nested-name-specifieropt class-name
class-virt-specifier:
    final
class-key:
    class
    struct
    union
```

A class-specifier whose class-head omits the class-head-name defines an unnamed class. [Note: An unnamed class thus can't be final. — end note]

- A class-name is inserted into the scope in which it is declared immediately after the class-name is seen. The class-name is also inserted into the scope of the class itself; this is known as the injected-class-name. For purposes of access checking, the injected-class-name is treated as if it were a public member name. A class-specifier is commonly referred to as a class definition. A class is considered defined after the closing brace of its class-specifier has been seen even though its member functions are in general not yet defined. The optional attribute-specifier-seq appertains to the class; the attributes in the attribute-specifier-seq are thereafter considered attributes of the class whenever it is named.
- <sup>3</sup> If a class is marked with the *class-virt-specifier* final and it appears as a *base-type-specifier* in a *base-clause* (Clause 10), the program is ill-formed.
- <sup>4</sup> Complete objects and member subobjects of class type shall have nonzero size.<sup>107</sup> [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]
- 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]
- 6 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).

Classes 204

<sup>107)</sup> Base class subobjects are not so constrained.

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. —  $end\ note$ ]

- <sup>7</sup> 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. <sup>108</sup>
- A standard-layout struct is a standard-layout class defined with the class-key struct or the class-key class. A standard-layout union is a standard-layout class defined with the class-key union.
- $^9$  [Note: Standard-layout classes are useful for communicating with code written in other programming languages. Their layout is specified in  $9.2. end \ note$ ]
- A POD struct<sup>109</sup> is a non-union class that is both a trivial class and a standard-layout class, and has no non-static data members of type non-POD struct, non-POD union (or array of such types). Similarly, a POD union is a union that is both a trivial class and a standard layout class, and has no non-static data members of type non-POD struct, non-POD union (or array of such types). A POD class is a class that is either a POD struct or a POD union.

```
[Example:
struct N {
                      // neither trivial nor standard-layout
  int i;
  int j;
  virtual ~N();
};
struct T {
                      // trivial but not standard-layout
  int i;
private:
  int j;
};
struct SL {
                      // standard-layout but not trivial
  int i;
  int j;
  ~SL();
};
struct POD {
                      // both trivial and standard-layout
  int i;
  int j;
};
```

Classes 205

<sup>108)</sup> 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).

<sup>109)</sup> The acronym POD stands for "plain old data".

```
— end example]
```

If a class-head-name contains a nested-name-specifier, the class-specifier shall refer to a class that was previously declared directly in the class or namespace to which the nested-name-specifier refers, or in an element of the inline namespace set (7.3.1) of that namespace (i.e., not merely inherited or introduced by a using-declaration), and the class-specifier shall appear in a namespace enclosing the previous declaration. In such cases, the nested-name-specifier of the class-head-name of the definition shall not begin with a decltype-specifier.

9.1 Class names [class.name]

1 A class definition introduces a new type. [Example:

declare an overloaded (Clause 13) function f() and not simply a single function f() twice. For the same reason,

```
struct S { int a; };
struct S { int a; };  // error, double definition
is ill-formed because it defines S twice. — end example]
```

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:

— end example] A declaration consisting solely of class-key identifier; is either a redeclaration of the name in the current scope or a forward declaration of the identifier as a class name. It introduces the class name into the current scope. [Example:

§ 9.1 206

```
struct s { int a; };
 void g() {
                                    // hide global struct s
   struct s;
                                    // with a block-scope declaration
                                    // refer to local struct s
   s* p;
   struct s { char* p; };
                                    // define local struct s
   struct s;
                                    // redeclaration, has no effect
— end example | [Note: Such declarations allow definition of classes that refer to each other. [Example:
 class Vector;
 class Matrix {
   // ...
   friend Vector operator*(const Matrix&, const Vector&);
 };
 class Vector {
   // ...
   friend Vector operator*(const Matrix&, const Vector&);
  Declaration of friends is described in 11.3, operator functions in 13.5. — end example ]—end note]
```

Declaration of friends is described in 11.3, operator functions in 13.5. — end example ] — end note ]

[Note: An elaborated-type-specifier (7.1.6.3) can also be used as a type-specifier as part of a declaration. It differs from a class declaration in that if a class of the elaborated name is in scope the elaborated name will refer to it. — end note ] [Example:

```
struct s { int a; };

void g(int s) {
   struct s* p = new struct s;  // global s
   p->a = s;  // parameter s
}
```

 $-end \ example$ 

<sup>4</sup> [Note: The declaration of a class name takes effect immediately after the identifier is seen in the class definition or elaborated-type-specifier. For example,

```
class A * A;
```

first specifies A to be the name of a class and then redefines it as the name of a pointer to an object of that class. This means that the elaborated form class A must be used to refer to the class. Such artistry with names can be confusing and is best avoided.  $-end\ note$ 

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

[class.mem]

```
member-specification: member-specification member-specification opt access-specifier: member-specification opt
```

§ 9.2 207

```
member-declaration:
       attribute-specifier-seq_{opt} decl-specifier-seq_{opt} member-declarator-list_{opt};
       function-definition; opt
       using-declaration
       static assert-declaration
       template-declaration
       alias-declaration
member-declarator-list:
       member-declarator
       member-declarator-list, member-declarator
member-declarator:
       declarator\ virt\text{-}specifier\text{-}seq_{opt}\ pure\text{-}specifier_{opt}
       declarator\ brace-or-equal-initializer_{opt}
       identifier_{opt} attribute-specifier-seq<sub>opt</sub>: constant-expression
virt\text{-}specifier\text{-}seq:
       virt-specifier
       virt-specifier-seq virt-specifier
virt-specifier:
       override
       final
pure-specifier:
```

- The member-specification in a class definition declares the full set of members of the class; no member can be added elsewhere. Members of a class are data members, member functions (9.3), nested types, and enumerators. Data members and member functions are static or non-static; see 9.4. Nested types are classes (9.1, 9.7) and enumerations (7.2) defined in the class, and arbitrary types declared as members by use of a typedef declaration (7.1.3). The enumerators of an unscoped enumeration (7.2) defined in the class are members of the class. Except when used to declare friends (11.3) or to introduce the name of a member of a base class into a derived class (7.3.3), 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.
- <sup>3</sup> [Note: A single name can denote several function members provided their types are sufficiently different (Clause 13). end note]
- <sup>4</sup> 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]
- <sup>5</sup> 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).
- 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.
- <sup>7</sup> The decl-specifier-seq may be omitted in constructor, destructor, and conversion function declarations only; when declaring another kind of member the decl-specifier-seq shall contain a type-specifier that is not a cv-qualifier. The member-declarator-list can be omitted only after a class-specifier or an enum-specifier or in a friend declaration (11.3). A pure-specifier shall be used only in the declaration of a virtual function (10.3).
- The optional attribute-specifier-seq 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.

§ 9.2

<sup>9</sup> A virt-specifier-seq shall contain at most one of each virt-specifier. A virt-specifier-seq shall appear only in the declaration of a virtual member function (10.3).

- 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]
- 12 [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]
- 13 [Example: A simple example of a class definition is

```
struct tnode {
  char tword[20];
  int count;
  tnode *left;
  tnode *right;
};
```

which contains an array of twenty characters, an integer, and two pointers to objects of the same type. Once this definition has been given, the declaration

```
tnode s, *sp;
```

declares s to be a tnode and sp to be a pointer to a tnode. With these declarations, sp->count refers to the count member of the object to which sp points; s.left refers to the left subtree pointer of the object s; and s.right->tword[0] refers to the initial character of the tword member of the right subtree of s. — end example]

- Nonstatic data members of a (non-union) class with the same access control (Clause 11) are allocated so that later members have higher addresses within a class object. The order of allocation of non-static data members with different access control is unspecified (11). Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions (10.3) and virtual base classes (10.1).
- 15 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 unscoped enumerated type; and
  - every member of every anonymous union that is a member of class T.
- <sup>16</sup> In addition, if class T has a user-declared constructor (12.1), every non-static data member of class T shall have a name different from T.
- Two 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).
- <sup>18</sup> 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).
- 19 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

§ 9.2

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.

A pointer to a standard-layout struct object, suitably converted using a reinterpret\_cast, points to its initial member (or if that member is a bit-field, then to the unit in which it resides) and vice versa. [Note: There might therefore be unnamed padding within a standard-layout struct object, but not at its beginning, as necessary to achieve appropriate alignment. —end note]

### 9.3 Member functions

[class.mfct]

- Functions declared in the definition of a class, excluding those declared with a friend specifier (11.3), are called member functions of that class. A member function may be declared static in which case it is a *static* member function of its class (9.4); otherwise it is a *non-static* member function of its class (9.3.1, 9.3.2).
- <sup>2</sup> A member function may be defined (8.4) in its class definition, in which case it is an *inline* member function (7.1.2), or it may be defined outside of its class definition if it has already been declared but not defined in its class definition. A member function definition that appears outside of the class definition shall appear in a namespace scope enclosing the class definition. Except for member function definitions that appear outside of a class definition, and except for explicit specializations of member functions of class templates and member function templates (14.7) appearing outside of the class definition, a member function shall not be redeclared.
- <sup>3</sup> An inline member function (whether static or non-static) may also be defined outside of its class definition provided either its declaration in the class definition or its definition outside of the class definition declares the function as inline. [Note: Member functions of a class in namespace scope have external linkage. Member functions of a local class (9.8) have no linkage. See 3.5. end note]
- There shall be at most one definition of a non-inline member function in a program; no diagnostic is required. There may be more than one inline member function definition in a program. See 3.2 and 7.1.2.
- If the definition of a member function is lexically outside its class definition, the member function name shall be qualified by its class name using the :: operator. [Note: A name used in a member function definition (that is, in the parameter-declaration-clause including the default arguments (8.3.6) or in the member function body) is looked up as described in 3.4. end note] [Example:

```
struct X {
  typedef int T;
  static T count;
  void f(T);
};
void X::f(T t = count) { }
```

The member function f of class X is defined in global scope; the notation X::f specifies that the function f is a member of class X and in the scope of class X. In the function definition, the parameter type T refers to the typedef member T declared in class X and the default argument count refers to the static data member count declared in class X. —  $end\ example$ 

- <sup>6</sup> A static local variable in a member function always refers to the same object, whether or not the member function is inline.
- 7 Previously declared member functions may be mentioned in friend declarations.
- 8 Member functions of a local class shall be defined inline in their class definition, if they are defined at all.
- <sup>9</sup> [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,

§ 9.3

### 9.3.1 Nonstatic member functions

# [class.mfct.non-static]

- A non-static member function may be called for an object of its class type, or for an object of a class derived (Clause 10) from its class type, using the class member access syntax (5.2.5, 13.3.1.1). A non-static member function may also be called directly using the function call syntax (5.2.2, 13.3.1.1) from within the body of a member function of its class or of a class derived from its class.
- If a non-static member function of a class X is called for an object that is not of type X, or of a type derived from X, the behavior is undefined.
- When an *id-expression* (5.1) that is not part of a class member access syntax (5.2.5) and not used to form a pointer to member (5.3.1) is used in a member of class X in a context where this can be used (5.1.1), if name lookup (3.4) resolves the name in the *id-expression* to a non-static non-type member of some class C, and if either the *id-expression* is potentially evaluated or C is X or a base class of X, the *id-expression* is transformed into a class member access expression (5.2.5) using (\*this) (9.3.2) as the *postfix-expression* to the left of the . operator. [Note: If C is not X or a base class of X, the class member access expression is ill-formed. end note] Similarly during name lookup, when an unqualified-id (5.1) used in the definition of a member function for class X resolves to a static member, an enumerator or a nested type of class X or of a base class of X, the unqualified-id is transformed into a qualified-id (5.1) in which the nested-name-specifier names the class of the member function. [Example:

```
struct tnode {
  char tword[20];
  int count;
  tnode *left;
  tnode *right;
  void set(const char*, tnode* 1, tnode* r);
void tnode::set(const char* w, tnode* 1, tnode* r) {
  count = strlen(w)+1;
  if (sizeof(tword)<=count)</pre>
      perror("tnode string too long");
  strcpy(tword,w);
  left = 1;
  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]

§ 9.3.1 211

<sup>110)</sup> See, for example,  $\langle cstring \rangle$  (21.7).

<sup>4</sup> A non-static member function may be declared const, volatile, or const volatile. These *cv-qualifiers* affect the type of the this pointer (9.3.2). They also affect the function type (8.3.5) of the member function; a member function declared const is a *const* member function, a member function declared volatile is a *volatile* member function and a member function declared const volatile is a *const volatile* member function. [*Example*:

```
struct X {
  void g() const;
  void h() const volatile;
};
```

X::g is a const member function and X::h is a const volatile member function. — end example]

- A non-static member function may be declared with a ref-qualifier (8.3.5); see 13.3.1.
- <sup>6</sup> A non-static member function may be declared virtual (10.3) or pure virtual (10.4).

## 9.3.2 The this pointer

[class.this]

- 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\*.
- <sup>2</sup> 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++; }
  int h() const { return a++; } // error
};
int s::f() const { return a; }
```

The a++ in the body of s::h is ill-formed because it tries to modify (a part of) the object for which s::h() is called. This is not allowed in a const member function because this is a pointer to const; that is, \*this has const type. — end example]

- 3 Similarly, volatile semantics (7.1.6.1) apply in volatile member functions when accessing the object and its non-static data members.
- A cv-qualified member function can be called on an object-expression (5.2.5) only if the object-expression is as cv-qualified or less-cv-qualified than the member function. [Example:

```
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

<sup>5</sup> 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)

§ 9.3.2 212

```
and (12.4). — end note]
```

### 9.4 Static members

[class.static]

A data or function member of a class may be declared static in a class definition, in which case it is a static member of the class.

A static member s of class X may be referred to using the *qualified-id* expression X::s; it is not necessary to use the class member access syntax (5.2.5) to refer to a static member. A static member may be referred to using the class member access syntax, in which case the object expression is evaluated. [Example:

— end example]

A static member may be referred to directly in the scope of its class or in the scope of a class derived (Clause 10) from its class; in this case, the static member is referred to as if a qualified-id expression was used, with the nested-name-specifier of the qualified-id naming the class scope from which the static member is referenced. [Example:

```
int g();
struct X {
    static int g();
};
struct Y : X {
    static int i;
};
int Y::i = g();
    // equivalent to Y::g();
```

- end example]

- <sup>4</sup> 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

[class.static.mfct]

- <sup>1</sup> [Note: The rules described in 9.3 apply to static member functions. end note]
- [Note: A static member function does not have a this pointer (9.3.2). end note] A static member function shall not be virtual. There shall not be a static and a non-static member function with the same name and the same parameter types (13.1). A static member function shall not be declared const, volatile, or const volatile.

#### 9.4.2 Static data members

[class.static.data]

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

§ 9.4.2

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:

```
class process {
   static process* run_chain;
   static process* running;
};

process* process::running = get_main();
process* process::run_chain = running;
```

The static data member run\_chain of class process is defined in global scope; the notation process ::run\_chain specifies that the member run\_chain is a member of class process and in the scope of class process. In the static data member definition, the *initializer* expression refers to the static data member running of class process. — end example]

[Note: Once the static data member has been defined, it exists even if no objects of its class have been created. [Example: in the example above, run\_chain and running exist even if no objects of class process are created by the program. — end example ] — end note ]

- If a non-volatile const static data member is of integral or enumeration type, its declaration in the class definition can specify a brace-or-equal-initializer in which every initializer-clause that is an assignment-expression is a constant expression (5.19). A static data member of literal type can be declared in the class definition with the constexpr specifier; if so, its declaration shall specify a brace-or-equal-initializer in which every initializer-clause that is an assignment-expression is a constant expression. [Note: In both these cases, the member may appear in constant expressions. —end note] The member shall still be defined in a namespace scope if it is odr-used (3.2) in the program and the namespace scope definition shall not contain an initializer.
- <sup>4</sup> [Note: There shall be exactly one definition of a static data member that is odr-used (3.2) in a program; no diagnostic is required. end note] Unnamed classes and classes contained directly or indirectly within unnamed classes shall not contain static data members.
- <sup>5</sup> Static data members of a class in namespace scope have external linkage (3.5). A local class shall not have static data members.
- <sup>6</sup> Static data members are initialized and destroyed exactly like non-local variables (3.6.2, 3.6.3).
- <sup>7</sup> A static data member shall not be mutable (7.1.1).

## 9.5 Unions [class.union]

- 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

 $\S~9.5$ 

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. —  $end\ note$ 

<sup>3</sup> [Example: Consider the following union:

```
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:

```
u.m.~M();
new (&u.n) N;

— end example]
A union of the form
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

For the purpose of name lookup, after the anonymous union definition, the members of the anonymous union are considered to have been defined in the scope in which the anonymous union is declared. [Example:

```
void f() {
  union { int a; const char* p; };
  a = 1;
  p = "Jennifer";
}
```

Here a and p are used like ordinary (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.
- <sup>7</sup> A union for which objects or pointers are declared is not an anonymous union. [Example:

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). —  $end\ note$  ]

<sup>8</sup> 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;

 $\S~9.5~$  215

otherwise, its variant members are the non-static data members of all anonymous unions that are members of X.

9.6 Bit-fields [class.bit]

A member-declarator of the form

 $identifier_{opt}$   $attribute-specifier-seq_{opt}$ : constant-expression

specifies a bit-field; its length is set off from the bit-field name by a colon. The optional attribute-specifier-seq appertains to the entity being declared. The bit-field attribute is not part of the type of the class member. The constant-expression shall be an integral constant expression with a value greater than or equal to zero. The value of the integral constant expression may be larger than the number of bits in the object representation (3.9) of the bit-field's type; in such cases the extra bits are used as padding bits and do not participate in the value representation (3.9) of the bit-field. Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit. [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, long, or long 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 Ivalue 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:

### 9.7 Nested class declarations

[class.nest]

A class can be declared within another class. A class declared within another is called a *nested* class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. [Note: See 5.1 for restrictions on the use of non-static data members and non-static member functions. — end note]

[ Example: int x;

§ 9.7 216

```
int y;
 struct enclose {
   int x;
   static int s;
   struct inner {
     void f(int i) {
                                   // OK: operand of size of is an unevaluated operand
       int a = sizeof(x);
       x = i;
                                    // error: assign to enclose::x
       s = i;
                                    // OK: assign to enclose::s
                                    // OK: assign to global x
       ::x = i;
                                    // OK: assign to global y
       у
     void g(enclose* p, int i) {
                                    // OK: assign to enclose::x
       p->x = i;
   };
 };
 inner* p = 0;
                                    // error: inner not in scope
— end example]
```

<sup>2</sup> Member functions and static data members of a nested class can be defined in a namespace scope enclosing the definition of their class. [Example:

```
struct enclose {
   struct inner {
      static int x;
      void f(int i);
   };
};
int enclose::inner::x = 1;

void enclose::inner::f(int i) { /* ... */ }

— end example]
```

<sup>3</sup> 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:

<sup>4</sup> Like a member function, a friend function (11.3) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (9.4), but it has no special access rights to members of an enclosing class.

### 9.8 Local class declarations

[class.local]

A class can be declared within a function definition; such a class is called a *local* class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope, and has the same

§ 9.8 217

access to names outside the function as does the enclosing function. Declarations in a local class shall not odr-use (3.2) a variable with automatic storage duration from an enclosing scope. [Example:

```
int x;
 void f() {
   static int s ;
   int x;
   const int N = 5;
   extern int q();
   struct local {
                                   // error: odr-use of automatic variable x
     int g() { return x; }
                                   // OK
     int h() { return s; }
                                   // OK
     int k() { return ::x; }
                                   // OK
     int 1() { return q(); }
                                   // OK: not an odr-use
     int m() { return N; }
     int *n() { return &N; }
                                   // error: odr-use of automatic variable N
   };
 local* p = 0;
                                   // error: local not in scope
— end example]
```

- <sup>2</sup> 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.
- <sup>3</sup> 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.
- <sup>4</sup> A local class shall not have static data members.

### 9.9 Nested type names

[class.nested.type]

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:

§ 9.9 218

# 10 Derived classes

# [class.derived]

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 . . . ont
base-specifier:
       attribute-specifier-seq_{opt} base-type-specifier
       attribute-specifier-seq_{opt}virtual access-specifier_{opt} base-type-specifier
       attribute-specifier-seq_{opt} access-specifier {\tt virtual}_{opt} base-type-specifier
class-or-decltype:
       nested-name-specifier_{opt} class-name
       decltype-specifier
base-type-specifier:
       class-or-decltype
access-specifier:
      private
      protected
      public
```

The optional attribute-specifier-seq appertains to the base-specifier.

- 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 Ivalue of a derived class type can be bound to a reference to an accessible unambiguous base class (8.5.3). end note]
- <sup>3</sup> The base-specifier-list specifies the type of the base class subobjects contained in an object of the derived class type. [Example:

```
struct Base {
  int a, b, c;
};

struct Derived : Base {
  int b;
};

struct Derived2 : Derived {
```

Derived classes 219

```
int c;
};
```

Here, an object of class Perived2 will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which in turn will have a subobject of class Perived2 which is the subobject of class Perived2 which Perived2 which Perived2 which

- 4 A base-specifier followed by an ellipsis is a pack expansion (14.5.3).
- The order in which the base class subobjects are allocated in the most derived object (1.8) is unspecified. [Note: a derived class and its base class subobjects can be represented by a directed acyclic graph (DAG) where an arrow means "directly derived from." A DAG of subobjects is often referred to as a "subobject lattice."



Figure 2 — Directed acyclic graph

- <sup>6</sup> The arrows need not have a physical representation in memory. -end note
- <sup>7</sup> [Note: Initialization of objects representing base classes can be specified in constructors; see 12.6.2. end note]
- <sup>8</sup> [Note: A base class subobject might have a layout (3.7) different from the layout of a most derived object of the same type. A base class subobject might have a polymorphic behavior (12.7) different from the polymorphic behavior of a most derived object of the same type. A base class subobject may be of zero size (Clause 9); however, two subobjects that have the same class type and that belong to the same most derived object must not be allocated at the same address (5.10). end note]

### 10.1 Multiple base classes

[class.mi]

A class can be derived from any number of base classes. [Note: The use of more than one direct base class is often called multiple inheritance. —end note] [Example:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class D : public A, public B, public C { /* ... */ };
```

- end example]
- <sup>2</sup> [Note: The order of derivation is not significant except as specified by the semantics of initialization by constructor (12.6.2), cleanup (12.4), and storage layout (9.2, 11.1). end note]
- A class shall not be specified as a direct base class of a derived class more than once. [Note: A class can be an indirect base class more than once and can be a direct and an indirect base class. There are limited things that can be done with such a class. The non-static data members and member functions of the direct base class cannot be referred to in the scope of the derived class. However, the static members, enumerations and types can be unambiguously referred to. end note] [Example:

```
class X { /* \dots */ }; class Y : public X, public X { /* \dots */ }; // ill-formed
```

```
class L { public: int next; /* ... */ };
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { void f(); /* ... */ }; // well-formed
class D : public A, public L { void f(); /* ... */ }; // well-formed

— end example]
```

A base class specifier that does not contain the keyword virtual, specifies a non-virtual base class. A base class specifier that contains the keyword virtual, specifies a virtual base class. For each distinct occurrence of a non-virtual base class in the class lattice of the most derived class, the most derived object (1.8) shall contain a corresponding distinct base class subobject of that type. For each distinct base class that is specified virtual, the most derived object shall contain a single base class subobject of that type. [Example: for an object of class type C, each distinct occurrence of a (non-virtual) base class L in the class lattice of C corresponds one-to-one with a distinct L subobject within the object of type C. Given the class C defined above, an object of class C will have two subobjects of class L as shown below.



Figure 3 — Non-virtual base

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:

```
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).

<sup>6</sup> For another example,

```
class V { /* \dots */ };
class A : virtual public V { /* \dots */ };
class B : virtual public V { /* \dots */ };
class C : public A, public B { /* \dots */ };
```

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.

<sup>7</sup> A class can have both virtual and non-virtual base classes of a given type.

```
class B { /* \dots */ };
class X : virtual public B { /* \dots */ };
class Y : virtual public B { /* \dots */ };
class Z : public B { /* \dots */ };
class AA : public X, public Y, public Z { /* \dots */ };
```

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.



Figure 4 — Virtual base

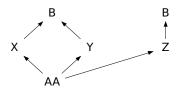


Figure 5 — Virtual and non-virtual base

Given the class AA defined above, class AA has two subobjects of class B: Z's B and the virtual B shared by X and Y, as shown below.

— end example]

# 10.2 Member name lookup

# [class.member.lookup]

- 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).
- <sup>2</sup> The following steps define the result of name lookup for a member name f in a class scope C.
- The lookup set for f in C, called S(f,C), consists of two component sets: the declaration set, a set of members named f; and the subobject set, a set of subobjects where declarations of these members (possibly including using-declarations) were found. In the declaration set, using-declarations are replaced by the members they designate, and type declarations (including injected-class-names) are replaced by the types they designate. S(f,C) is calculated as follows:
- If C contains a declaration of the name f, the declaration set contains every declaration of f declared in C that satisfies the requirements of the language construct in which the lookup occurs. [Note: Looking up a name in an elaborated-type-specifier (3.4.4) or base-specifier (Clause 10), for instance, ignores all non-type declarations, while looking up a name in a nested-name-specifier (3.4.3) ignores function, 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.
- 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).
- <sup>6</sup> 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.
- <sup>7</sup> 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:

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

<sup>8</sup> If the name of an overloaded function is unambiguously found, overloading resolution (13.3) also takes place before access control. Ambiguities can often be resolved by qualifying a name with its class name. [Example:

```
struct A {
   int f();
};

struct B {
   int f();
};

struct C : A, B {
   int f() { return A::f() + B::f(); }
};
```

— end example]

[Note: A static member, a nested type or an enumerator defined in a base class T can unambiguously be found even if an object has more than one base class subobject of type T. Two base class subobjects share the non-static member subobjects of their common virtual base classes. — end note] [Example:

```
struct V {
   int v;
};
struct A {
   int a;
   static int s;
   enum { e };
```

[Note: When virtual base classes are used, a hidden declaration can be reached along a path through the subobject lattice that does not pass through the hiding declaration. This is not an ambiguity. The identical use with non-virtual base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others. — end note ] [Example:

```
struct V { int f(); int x; };
struct W { int g(); int y; };
struct B : virtual V, W {
  int f(); int x;
  int g(); int y;
};
struct C : virtual V, W { };
```

— end example]

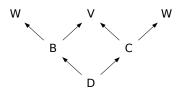


Figure 6 — Name lookup

<sup>11</sup> [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]

— end example]

An explicit or implicit conversion from a pointer to or an expression designating an object of a derived class to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing the base class. [Example:

```
struct V { };
```

[Note: Even if the result of name lookup is unambiguous, use of a name found in multiple subobjects might still be ambiguous (4.11, 5.2.5, 11.2). — end note] [Example:

```
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
                                   // Unambiguous (static)
     f(0);
                                   // Unambiguous (only one B2)
     f(0.0);
                                   // Unambiguous
     int B1::* mpB1 = &D::i;
     int D::* mpD = &D::i;
                                   // Ambiguous conversion
   }
 };
— end example]
```

## 10.3 Virtual functions

 $-end \ example$ 

[class.virtual]

Virtual functions support dynamic binding and object-oriented programming. A class that declares or inherits a virtual function is called a *polymorphic class*.

If a virtual member function vf is declared in a class Base and in a class Derived, derived directly or indirectly from Base, a member function vf with the same name, parameter-type-list (8.3.5), cv-qualification, and ref-qualifier (or absence of same) as Base::vf is declared, then Derived::vf is also virtual (whether or not it is so declared) and it overrides<sup>111</sup> Base::vf. For convenience we say that any virtual function overrides itself. A virtual member function C::vf of a class object S is a final overrider unless the most derived class (1.8) of which S is a base class subobject (if any) declares or inherits another member function that overrides vf. In a derived class, if a virtual member function of a base class subobject has more than one final overrider the program is ill-formed. [Example:

<sup>111)</sup> 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 virtual specifier in the declaration of an overriding function is legal but redundant (has empty semantics). Access control (Clause 11) is not considered in determining overriding.

```
struct A {
       virtual void f();
     };
     struct B : virtual A {
       virtual void f();
     };
     struct C : B , virtual A {
       using A::f;
     };
     void foo() {
       Cc;
                             // calls B::f, the final overrider
       c.f();
       c.C::f();
                             // calls A::f because of the using-declaration
   — end example]
      [Example:
     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]
<sup>3</sup> [Note: A virtual member function does not have to be visible to be overridden, for example,
     struct B {
       virtual void f();
     };
     struct D : B {
       void f(int);
     };
     struct D2 : D {
       void f();
     };
   the function f(int) in class D hides the virtual function f() in its base class B; D::f(int) is not a virtual
   function. However, f() declared in class D2 has the same name and the same parameter list as B::f(), and
   therefore is a virtual function that overrides the function B::f() even though B::f() is not visible in class
   D2. -end note
  If a virtual function f in some class B is marked with the virt-specifier final and in a class D derived from
   B a function D::f overrides B::f, the program is ill-formed. [Example:
     struct B {
       virtual void f() const final;
     };
```

<sup>5</sup> If a virtual function is marked with the *virt-specifier* override and does not override a member function of a base class, the program is ill-formed. [*Example*:

// error: D::f attempts to override final B::f

struct D : B {
 void f() const;

— end example]

**}**;

```
struct B {
  virtual void f(int);
};

struct D : B {
  virtual void f(long) override;  // error: wrong signature overriding B::f
  virtual void f(int) override;  // OK
};
```

— end example]

<sup>6</sup> 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.

- <sup>7</sup> 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  $^{112}$
  - 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.
- If the return type of D::f differs from the return type of B::f, the class type in the return type of D::f shall be complete at the point of declaration of D::f or shall be the class type D. When the overriding function is called as the final overrider of the overridden function, its result is converted to the type returned by the (statically chosen) overridden function (5.2.2). [Example:

```
class B { };
class D : private B { friend class Derived; };
struct Base {
  virtual void vf1();
  virtual void vf2();
  virtual void vf3();
  virtual B*
                vf4();
  virtual B*
                vf5();
  void f();
};
struct No_good : public Base {
  D* vf4();
                   // error: B (base class of D) inaccessible
};
class A;
struct Derived : public Base {
                     // virtual and overrides Base::vf1()
    void vf1();
    void vf2(int); // not virtual, hides Base::vf2()
                     // error: invalid difference in return type only
    char vf3();
                     // OK: returns pointer to derived class
    D* vf4();
                     // error: returns pointer to incomplete class
    A* vf5();
    void f();
};
```

<sup>112)</sup> Multi-level pointers to classes or references to multi-level pointers to classes are not allowed.

```
void g() {
  Derived d:
                                   // standard conversion:
  Base* bp = \&d;
                                    // Derived* to Base*
                                   // calls Derived::vf1()
  bp->vf1();
  bp->vf2();
                                   // calls Base::vf2()
  bp->f();
                                   // calls Base::f() (not virtual)
                                   // calls Derived::pf() and converts the
  B* p = bp->vf4();
                                   // result to B*
  Derived* dp = &d;
                                   // calls Derived::pf() and does not
  D* q = dp - vf4();
                                   // convert the result to B*
  dp->vf2();
                                   // ill-formed: argument mismatch
}
```

— end example]

<sup>9</sup> [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]

- [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
- A virtual function declared in a class shall be defined, or declared pure (10.4) in that class, or both; but no diagnostic is required (3.2).
- 12 [Example: here are some uses of virtual functions with multiple base classes:

```
struct A {
  virtual void f();
};
struct B1 : A {
                                   // note non-virtual derivation
  void f();
};
struct B2 : A {
  void f();
};
struct D : B1, B2 {
                                   // D has two separate A subobjects
};
void foo() {
// A* ap = &d; // would be ill-formed: ambiguous
  B1* b1p = &d;
       ap = b1p;
  D*
       dp = \&d;
                                   // calls D::B1::f
  ap->f();
                                   // ill-formed: ambiguous
  dp->f();
```

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.

13 The following example shows a function that does not have a unique final overrider:

```
struct A {
   virtual void f();
};

struct VB1 : virtual A {
   void f();
};

struct VB2 : virtual A {
   void f();
};

struct Error : VB1, VB2 {
   // ill-formed
};

struct Okay : VB1, VB2 {
   void f();
};
```

Both VB1::f and VB2::f override A::f but there is no overrider of both of them in class Error. This example is therefore ill-formed. Class Okay is well formed, however, because Okay::f is a final overrider.

The following example uses the well-formed classes from above.

```
struct VB1a : virtual A {
};

struct Da : VB1a, VB2 {
};

void foe() {
   VB1a* vb1ap = new Da;
   vb1ap->f();
}
// calls VB2::f
```

- end example]

<sup>15</sup> Explicit qualification with the scope operator (5.1) suppresses the virtual call mechanism. [Example:

```
class B { public: virtual void f(); };
class D : public B { public: void f(); };
void D::f() { /* ... */ B::f(); }
```

Here, the function call in D::f really does call B::f and not D::f. — end example

A function with a deleted definition (8.4) shall not override a function that does not have a deleted definition. Likewise, a function that does not have a deleted definition shall not override a function with a deleted definition.

### 10.4 Abstract classes

[class.abstract]

- <sup>1</sup> The abstract class mechanism supports the notion of a general concept, such as a **shape**, of which only more concrete variants, such as **circle** and **square**, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations.
- An abstract class is a class that can be used only as a base class of some other class; no objects of an abstract class can be created except as subobjects of a class derived from it. A class is abstract if it has at least one pure virtual function. [Note: Such a function might be inherited: see below. —end note] A virtual function is specified pure by using a pure-specifier (9.2) in the function declaration in the class definition. A

pure virtual function need be defined only if called with, or as if with (12.4), the qualified-id syntax (5.1). [Example:

```
class point { /* ... */ };
  class shape {
                                    // abstract class
    point center;
  public:
    point where() { return center; }
    void move(point p) { center=p; draw(); }
    virtual void rotate(int) = 0; // pure virtual
    virtual void draw() = 0;
                                   // pure virtual
  };
— end example ] [Note: A function declaration cannot provide both a pure-specifier and a definition — end
note | [Example:
  struct C {
                                   // ill-formed
    virtual void f() = 0 { };
— end example]
```

<sup>3</sup> 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:

— end example]

<sup>4</sup> A class is abstract if it contains or inherits at least one pure virtual function for which the final overrider is pure virtual. [ *Example*:

```
class ab_circle : public shape {
  int radius;
public:
  void rotate(int) { }
  // ab_circle::draw() is a pure virtual
}.
```

Since shape::draw() is a pure virtual function ab\_circle::draw() is a pure virtual by default. The alternative declaration,

would make class circle nonabstract and a definition of circle::draw() must be provided. —  $end\ example$ ]

- <sup>5</sup> [Note: An abstract class can be derived from a class that is not abstract, and a pure virtual function may override a virtual function which is not pure.  $end\ note$ ]
- Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (10.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.

# 11 Member access control [class.access]

- 1 A member of a class can be
  - 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 11.4).
  - public; that is, its name can be used anywhere without access restriction.
- <sup>2</sup> 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. <sup>113</sup>
- 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:

— end example]

Access control is applied uniformly to all names, whether the names are referred to from declarations or expressions. [Note: Access control applies to names nominated by friend declarations (11.3) and using-declarations (7.3.3). — end note] In the case of overloaded function names, access control is applied to the function selected by overload resolution. [Note: Because access control applies to names, if access control is applied to a typedef name, only the accessibility of the typedef name itself is considered. The accessibility of the entity referred to by the typedef is not considered. For example,

- -end note
- <sup>5</sup> It should be noted that it is *access* to members and base classes that is controlled, not their *visibility*. Names of members are still visible, and implicit conversions to base classes are still considered, when those members and base classes are inaccessible. The interpretation of a given construct is established without regard to access control. If the interpretation established makes use of inaccessible member names or base classes, the construct is ill-formed.
- <sup>6</sup> All access controls in Clause 11 affect the ability to access a class member name from the declaration of a particular entity, including parts of the declaration preceding the name of the entity being declared and, if the

Member access control 231

<sup>113)</sup> Access permissions are thus transitive and cumulative to nested and local classes.

entity is a class, the definitions of members of the class appearing outside the class's member-specification. [Note: this access also applies to implicit references to constructors, conversion functions, and destructors. — end note] [Example:

```
class A {
  typedef int I;
                    // private member
  I f();
  friend I g(I);
  static I x;
  template<int> struct Q;
  template<int> friend struct R;
protected:
    struct B { };
};
A::I A::f() { return 0; }
A::I g(A::I p = A::x);
A::I g(A::I p) { return 0; }
A::I \ A::x = 0;
template<A::I> struct A::Q { };
template<A::I> struct R { };
struct D: A::B, A { };
```

- Here, all the uses of A::I are well-formed because A::f, A::x, and A::Q are members of class A and g and R are friends of class A. This implies, for example, that access checking on the first use of A::I must be deferred until it is determined that this use of A::I is as the return type of a member of class A. Similarly, the use of A::B as a base-specifier is well-formed because D is derived from A, so checking of base-specifiers must be deferred until the entire base-specifier-list has been seen. end example
- 8 The names in a default argument (8.3.6) are bound at the point of declaration, and access is checked at that point rather than at any points of use of the default argument. Access checking for default arguments in function templates and in member functions of class templates is performed as described in 14.7.1.
- 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:

```
class B { };
template <class T> class C {
protected:
   typedef T TT;
};

template <class U, class V = typename U::TT>
class D : public U { };

D <C<B> >* d;  // access error, C::TT is protected

— end example]
```

### 11.1 Access specifiers

[class.access.spec]

Member declarations can be labeled by an access-specifier (Clause 10):

```
access-specifier: member-specification_{opt}
```

An access-specifier specifies the access rules for members following it until the end of the class or until another access-specifier is encountered. [Example:

§ 11.1 232

<sup>2</sup> Any number of access specifiers is allowed and no particular order is required. [Example:

— end example]

- <sup>3</sup> [Note: The effect of access control on the order of allocation of data members is described in 9.2. end note]
- When a member is redeclared within its class definition, the access specified at its redeclaration shall be the same as at its initial declaration. [Example:

— end example]

[Note: In a derived class, the lookup of a base class name will find the injected-class-name instead of the name of the base class in the scope in which it was declared. The injected-class-name might be less accessible than the name of the base class in the scope in which it was declared. —end note]

[Example:

— end example]

# 11.2 Accessibility of base classes and base class members

[class.access.base]

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.

§ 11.2 233

<sup>114)</sup> 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.

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:

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. —  $end\ example$ ]

<sup>3</sup> [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,

```
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;
                                    // OK (b.mi is different from this->mi)
   b.mi = 3;
   b.si = 3;
                                    // OK (b.si is different from this->si)
                                    // OK
   ::B::si = 3;
                                    // error: B is a private base class
   ::B* bp1 = this;
                                    // OK with cast
   ::B* bp2 = (::B*)this;
   bp2->mi = 3;
                                    // OK: access through a pointer to B.
-end note
```

- <sup>4</sup> 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:

§ 11.2 234

— end example]

- If a base class is accessible, one can implicitly convert a pointer to a derived class to a pointer to that base class (4.10, 4.11). [Note: It follows that members and friends of a class X can implicitly convert an X\* to a pointer to a private or protected immediate base class of X. end note] The access to a member is affected by the class in which the member is named. This naming class is the class in which the member name was looked up and found. [Note: This class can be explicit, e.g., when a qualified-id is used, or implicit, e.g., when a class member access operator (5.2.5) is used (including cases where an implicit "this->" is added). If both a class member access operator and a qualified-id are used to name the member (as in p->T::m), the class naming the member is the class denoted by the nested-name-specifier of the qualified-id (that is, T). end note] A member m is accessible at the point R when named in class N if
  - m as a member of N is public, or
  - m as a member of N is private, and R occurs in a member or friend of class N, or
  - m as a member of N is protected, and R occurs in a member or friend of class N, or in a member or friend of a class P derived from N, where m as a member of P is public, private, or protected, or
  - there exists a base class B of N that is accessible at R, and m is accessible at R when named in class B. [Example:

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.

§ 11.2 235

[Note: This requirement is in addition to the requirement that the member be accessible as named. -end note]

11.3 Friends [class.friend]

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:

```
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]
```

[Example:

<sup>2</sup> Declaring a class to be a friend implies that the names of private and protected members from the class granting friendship can be accessed in the *base-specifiers* and member declarations of the befriended class.

```
class A {
   class B { };
   friend class X;
 };
                       // OK: A::B accessible to friend
 struct X : A::B {
                       // OK: A::B accessible to member of friend
   A::B mx;
   class Y {
     A::B my;
                       // OK: A::B accessible to nested member of friend
   };
 };
— end example ] [Example:
 class X {
   enum { a=100 };
   friend class Y;
 };
 class Y {
                       // OK, Y is a friend of X
   int v[X::a];
 };
 class Z {
   int v[X::a];
                       // error: X::a is private
 };
```

§ 11.3 236

```
— end example]
      A class shall not be defined in a friend declaration. [Example:
       friend class B { }; // error: cannot define class in friend declaration
     };
    - end example]
  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 {
                           // OK: class C is a friend
       friend C;
     };
     class X2 {
       friend Ct;
                           // OK: class C is a friend
                           // error: no type-name D in scope
       friend D;
       friend class D;
                           // OK: elaborated-type-specifier declares new class
     };
     template <typename T> class R {
       friend T;
     };
                           // class C is a friend of R<C>
     R<C> rc;
                           // OK: "friend int;" is ignored
     R<int> Ri;
    — end example]
4 A function first declared in a friend declaration has external linkage (3.5). Otherwise, the function retains
   its previous linkage (7.1.1).
<sup>5</sup> When a friend declaration refers to an overloaded name or operator, only the function specified by the
   parameter types becomes a friend. A member function of a class X can be a friend of a class Y. [Example:
     class Y {
       friend char* X::foo(int);
                                         // constructors can be friends
       friend X::X(char);
       friend X::~X();
                                         // destructors can be friends
     };
    — end example]
<sup>6</sup> A function can be defined in a friend declaration of a class if and only if the class is a non-local class (9.8),
   the function name is unqualified, and the function has namespace scope. [Example:
     class M {
                                         // definition of global f, a friend of M,
       friend void f() { }
                                         // not the definition of a member function
     };
   — end example]
   § 11.3
                                                                                                              237
```

<sup>7</sup> 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).

- 8 No storage-class-specifier shall appear in the decl-specifier-seq of a friend declaration.
- <sup>9</sup> 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.
- <sup>10</sup> Friendship is neither inherited nor transitive. [Example:

```
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) {
                      // error: D is not a friend of A
    p->a++;
                      // despite being derived from a friend
  }
};
```

— end example]

If a friend declaration appears in a local class (9.8) and the name specified is an unqualified name, a prior declaration is looked up without considering scopes that are outside the innermost enclosing non-class scope. For a friend function declaration, if there is no prior declaration, the program is ill-formed. For a friend class declaration, if there is no prior declaration, the class that is specified belongs to the innermost enclosing non-class scope, but if it is subsequently referenced, its name is not found by name lookup until a matching declaration is provided in the innermost enclosing nonclass scope. [Example:

```
class X;
void a();
void f() {
  class Y;
  extern void b();
  class A {
                     // OK, but X is a local class, not ::X
  friend class X;
                     // OK
  friend class Y;
                     // OK, introduces local class Z
  friend class Z;
  friend void a();
                     // error, ::a is not considered
  friend void b();
                     //OK
  friend void c();
                     // error
  };
                      // OK, but :: X is found
  X *px;
                     // error, no Z is found
  Z *pz;
}
```

§ 11.3

— end example]

#### 11.4 Protected member access

[class.protected]

An additional access check beyond those described earlier in Clause 11 is applied when a non-static data member or non-static member function is a protected member of its naming class (11.2)<sup>115</sup> As described earlier, access to a protected member is granted because the reference occurs in a friend or member of some class C. If the access is to form a pointer to member (5.3.1), the nested-name-specifier shall denote C or a class derived from C. All other accesses involve a (possibly implicit) object expression (5.2.5). In this case, the class of the object expression shall be C or a class derived from C. [Example:

```
class B {
protected:
  int i;
  static int j;
class D1 : public B {
class D2 : public B {
  friend void fr(B*,D1*,D2*);
  void mem(B*,D1*);
};
void fr(B* pb, D1* p1, D2* p2) {
                                  // ill-formed
  pb->i = 1;
                                  // ill-formed
  p1->i = 2;
                                  // OK (access through a D2)
  p2->i = 3;
                                  // OK (access through a D2, even though
  p2->B::i = 4;
                                  // naming class is B)
                                  // ill-formed
  int B::* pmi_B = &B::i;
                                  // OK (type of &D2::i is int B::*)
  int B::* pmi_B2 = &D2::i;
  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
                                  // ill-formed
  p1->i = 2;
                                  // OK (access through this)
  i = 3;
                                  // OK (access through this, qualification ignored)
  B::i = 4;
  int B::* pmi_B = &B::i;
                                  // ill-formed
  int B::* pmi_B2 = &D2::i;
                                  //OK
                                  // OK (because j refers to static member)
  i = 5;
  B::j = 6;
                                  // OK (because B::j refers to static member)
void g(B* pb, D1* p1, D2* p2) {
                                   // ill-formed
  pb->i = 1;
                                  // ill-formed
  p1->i = 2;
  p2->i = 3;
                                   // ill-formed
```

§ 11.4 239

<sup>115)</sup> This additional check does not apply to other members, e.g., static data members or enumerator member constants.

— end example]

#### 11.5 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() {
  Dd;
  B* pb = &d;
  D* pd = &d;
  pb->f();
                                   // OK: B::f() is public,
                                   // D::f() is invoked
                                   // error: D::f() is private
 pd->f();
```

-end example

<sup>2</sup> Access is checked at the call point using the type of the expression used to denote the object for which the member function is called (B\* in the example above). The access of the member function in the class in which it was defined (D in the example above) is in general not known.

# 11.6 Multiple access

[class.paths]

<sup>1</sup> If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives most access. [Example:

```
class W { public: void f(); };
class A : private virtual W { };
class B : public virtual W { };
class C : public A, public B {
  void f() { W::f(); }
};
```

<sup>2</sup> Since W::f() is available to C::f() along the public path through B, access is allowed. — end example]

# 11.7 Nested classes

[class.access.nest]

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:

§ 11.7 240

 $\odot$  ISO/IEC N3337

§ 11.7 241

# 12 Special member functions [special]

The default constructor (12.1), copy constructor and copy assignment operator (12.8), move constructor and move assignment operator (12.8), and destructor (12.4) are special member functions. [Note: The implementation will implicitly declare these member functions for some class types when the program does not explicitly declare them. The implementation will implicitly define them if they are odr-used (3.2). See 12.1, 12.4 and 12.8. — end note] Programs shall not define implicitly-declared special member functions.

Programs may explicitly refer to implicitly-declared special member functions. [Example: a program may explicitly call, take the address of or form a pointer to member to an implicitly-declared special member function.

- end example]
- <sup>3</sup> [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]
- <sup>4</sup> 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]

- Constructors do not have names. A special declarator syntax is used to declare or define the constructor. The syntax uses:
  - an optional decl-specifier-seq in which each decl-specifier is either a function-specifier or constexpr,
  - the constructor's class name, and
  - a parameter list

in that order. In such a declaration, optional parentheses around the constructor class name are ignored. [ Example:

- end example]
- A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (5.2.3) will cause a constructor to be called to initialize an object. [Note: For initialization of objects of class type see 12.6. end note]
- 3 A typedef-name shall not be used as the class-name in the declarator-id for a constructor declaration.

§ 12.1 242

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-static 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),
  - 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, or
  - any direct or virtual base class or non-static data member has a type with a destructor that is deleted or inaccessible from the defaulted default constructor.

A default constructor is trivial if it is not user-provided and if:

- 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 odrused (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]
- <sup>7</sup> 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

§ 12.1 243

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).

- <sup>8</sup> [Note: 12.6.2 describes the order in which constructors for base classes and non-static data members are called and describes how arguments can be specified for the calls to these constructors. end note]
- <sup>9</sup> 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 void) shall be specified for a constructor. A return statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.
- A functional notation type conversion (5.2.3) can be used to create new objects of its type. [Note: The syntax looks like an explicit call of the constructor. —end note] [Example:

```
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
```

- end example]
- An object created in this way is unnamed. [Note: 12.2 describes the lifetime of temporary objects. end note] [Note: Explicit constructor calls do not yield lvalues, see 3.10. end note]
- $^{13}$  [Note: some language constructs have special semantics when used during construction; see 12.6.2 and 12.7. end note]
- During the construction of a **const** object, if the value of the object or any of its subobjects is accessed through a glvalue that is not obtained, directly or indirectly, from the constructor's **this** pointer, the value of the object or subobject thus obtained is unspecified. [Example:

### 12.2 Temporary objects

[class.temporary]

Temporaries of class type are created in various contexts: binding a reference to a prvalue (8.5.3), returning a prvalue (6.6.3), a conversion that creates a prvalue (4.1, 5.2.9, 5.2.11, 5.4), throwing an exception (15.1), entering a handler (15.3), and in some initializations (8.5). [Note: The lifetime of exception objects is described in 15.1. — end note] Even when the creation of the temporary object is unevaluated (Clause 5) or otherwise avoided (12.8), all the semantic restrictions shall be respected as if the temporary object had been created and later destroyed. [Note: even if there is no call to the destructor or copy/move constructor, all the semantic restrictions, such as accessibility (Clause 11) and whether the function is deleted (8.4.3), shall be satisfied. However, in the special case of a function call used as the operand of a decltype-specifier (5.2.2), no temporary is introduced, so the foregoing does not apply to the prvalue of any such function call. — end note]

§ 12.2 244

<sup>2</sup> [Example: Consider the following code:

```
class X {
public:
  X(int);
  X(const X&);
  X& operator=(const X&);
  ~X():
};
class Y {
public:
  Y(int);
  Y(Y&&);
  ~Y();
};
X f(X);
Y g(Y);
void h() {
  X a(1);
  X b = f(X(2));
  Y c = g(Y(3));
   = f(a);
```

An implementation might use a temporary in which to construct X(2) before passing it to f() using X's copy constructor; alternatively, X(2) might be constructed in the space used to hold the argument. Likewise, an implementation might use a temporary in which to construct Y(3) before passing it to g() using Y's move constructor; alternatively, Y(3) might be constructed in the space used to hold the argument. Also, a temporary might be used to hold the result of f(X(2)) before copying it to b using X's copy constructor; alternatively, f()'s result might be constructed in b. Likewise, a temporary might be used to hold the result of g(Y(3)) before moving it to c using Y's move constructor; alternatively, g()'s result might be constructed in c. On the other hand, the expression a=f(a) requires a temporary for the result of f(a), which is then assigned to a. — end example]

- When an implementation introduces a temporary object of a class that has a non-trivial constructor (12.1, 12.8), it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary with a non-trivial destructor (12.4). Temporary objects are destroyed as the last step 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.
- <sup>4</sup> There are two contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array. If the constructor has one or more default arguments, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any.
- 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.

§ 12.2 245

— 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:

```
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:

```
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 [class.conv]

- <sup>1</sup> 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).
- <sup>2</sup> 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).
- <sup>3</sup> [Note: See 13.3 for a discussion of the use of conversions in function calls as well as examples below. -end note]

§ 12.3 246

<sup>4</sup> At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single value.

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:

#### 12.3.1 Conversion by constructor

[class.conv.ctor]

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:

— end example]

An explicit constructor constructs objects just like non-explicit constructors, but does so only where the direct-initialization syntax (8.5) or where casts (5.2.9, 5.4) are explicitly used. A default constructor

§ 12.3.1

may be an explicit constructor; such a constructor will be used to perform default-initialization or value-initialization (8.5). [Example:

```
struct Z {
   explicit Z();
   explicit Z(int);
 };
                                     // OK: default-initialization performed
 Za;
 Z a1 = 1;
                                     // error: no implicit conversion
                                     // OK: direct initialization syntax used
 Z = Z(1);
                                     // OK: direct initialization syntax used
 Z a2(1);
                                     // OK: direct initialization syntax used
 Z* p = new Z(1);
                                     // OK: explicit cast used
 Z a4 = (Z)1;
 Z = static_cast < Z > (1);
                                     // OK: explicit cast used
— end example]
```

<sup>3</sup> 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]

A member function of a class X having no parameters with a name of the form

```
conversion-function-id:
    operator conversion-type-id
conversion-type-id:
    type-specifier-seq conversion-declarator<sub>opt</sub>
conversion-declarator:
    ptr-operator conversion-declarator<sub>opt</sub>
```

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. <sup>116</sup>

```
Example:
struct X {
  operator int();
};

void f(X a) {
  int i = int(a);
  i = (int)a;
  i = a;
}
```

In all three cases the value assigned will be converted by X::operator int(). — end example]

<sup>2</sup> 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:

§ 12.3.2 248

<sup>116)</sup> 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.

— end example]

<sup>3</sup> 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:

The \* is the pointer declarator and not the multiplication operator. — end example ] — end note ]

- <sup>4</sup> Conversion functions are inherited.
- <sup>5</sup> Conversion functions can be virtual.
- <sup>6</sup> Conversion functions cannot be declared static.

# 12.4 Destructors [class.dtor]

- 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.
- A declaration of a destructor that does not have an *exception-specification* is implicitly considered to have the same *exception-specification* as an implicit declaration (15.4).
- <sup>4</sup> 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.
- <sup>5</sup> 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,
  - any direct or virtual base class has a deleted destructor or a destructor that is inaccessible from the defaulted destructor,

§ 12.4 249

— or, for a virtual destructor, lookup of the non-array deallocation function results in an ambiguity or in a function that is deleted or inaccessible from the defaulted destructor.

A destructor is trivial if it is not user-provided and if:

- 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.

- <sup>6</sup> A destructor that is defaulted and not defined as deleted is *implicitly defined* when it is odr-used (3.2) to destroy an object of its class type (3.7) or when it is explicitly defaulted after its first declaration.
- <sup>7</sup> Before the defaulted destructor for a class is implicitly defined, all the non-user-provided destructors for its base classes and its non-static data members shall have been implicitly defined.
- After executing the body of the destructor and destroying any automatic objects allocated within the body, a destructor for class X calls the destructors for X's direct non-variant non-static data members, the destructors for X's 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).
- <sup>9</sup> 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.
- <sup>10</sup> [Note: some language constructs have special semantics when used during destruction; see 12.7. end note]
- 11 Destructors are invoked implicitly
  - for constructed objects with static storage duration (3.7.1) at program termination (3.6.3),
  - 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]

§ 12.4 250

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:

```
struct B {
  virtual ~B() { }
};
struct D : B {
  ~D() { }
D D_object;
typedef B B_alias;
B* B_ptr = &D_object;
void f() {
                                   // calls B's destructor
  D_object.B::~B();
                                   // calls D's destructor
  B_ptr->~B();
                                   // calls D's destructor
  B_ptr->~B_alias();
                                   // calls B's destructor
  B_ptr->B_alias::~B();
  B_ptr->B_alias::~B_alias();
                                   // calls B's destructor
```

— end example] [Note: An explicit destructor call must always be written using a member access operator (5.2.5) or a qualified-id (5.1); in particular, the unary-expression "X() in a member function is not an explicit destructor call (5.3.1). — end note]

[Note: explicit calls of destructors are rarely needed. One use of such calls is for objects placed at specific addresses using a new-expression with the placement option. Such use of explicit placement and destruction of objects can be necessary to cope with dedicated hardware resources and for writing memory management facilities. For example,

-end note

Once a destructor is invoked for an object, the object no longer exists; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended (3.8). [Example: if the destructor for an automatic object is explicitly invoked, and the block is subsequently left in a manner that would ordinarily invoke implicit destruction of the object, the behavior is undefined. —end example]

<sup>6</sup> [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,

§ 12.4 251

```
typedef int I;
I* p;
p->I::~I();
— end note]
```

12.5 Free store [class.free]

Any allocation function for a class T is a static member (even if not explicitly declared static).

<sup>2</sup> [Example:

 $-end \ example$ 

- 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).
- <sup>4</sup> 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).<sup>117</sup> Otherwise, if the *delete-expression* is used to deallocate an object of class T or array thereof, the static and dynamic types of the object shall be identical and the deallocation function's name is looked up in the scope of T. If this lookup fails to find the name, the name is looked up in the global scope. If the result of the lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed.
- When a *delete-expression* is executed, the selected deallocation function shall be called with the address of the block of storage to be reclaimed as its first argument and (if the two-parameter style is used) the size of the block as its second argument.<sup>118</sup>
- Any deallocation function for a class X is a static member (even if not explicitly declared static). [Example:

```
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*);
};
```

<sup>117)</sup> A similar provision is not needed for the array version of operator delete because 5.3.5 requires that in this situation, the static type of the object to be deleted be the same as its dynamic type.

<sup>118)</sup> 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 5.3.5.

```
— end example]
```

7 Since member allocation and deallocation functions are **static** they cannot be virtual. [Note: however, when the cast-expression of a delete-expression refers to an object of class type, because the deallocation function actually called is looked up in the scope of the class that is the dynamic type of the object, if the destructor is virtual, the effect is the same. For example,

```
struct B {
  virtual ~B();
  void operator delete(void*, std::size_t);
};

struct D : B {
  void operator delete(void*);
};

void f() {
  B* bp = new D;
  delete bp;  //1: uses D::operator delete(void*)
}
```

Here, storage for the non-array object of class D is deallocated by D::operator delete(), due to the virtual destructor. —  $end\ note$ ] [Note: Virtual destructors have no effect on the deallocation function actually called when the cast-expression of a delete-expression refers to an array of objects of class type. For example,

— end note]

- <sup>8</sup> 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]
- 9 [Note: If a deallocation function has no explicit exception-specification, it is treated as if it were specified with noexcept(true) (15.4). end note]

12.6 Initialization [class.init]

- 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.
- An object of class type (or array thereof) can be explicitly initialized; see 12.6.1 and 12.6.2.
- 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;

§ 12.6 253

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]

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
                                   // call sqrt(complex,complex)
complex d = sqrt(b,c);
                                   // and copy/move the result into d
complex e;
                                   // initialize by a call of
                                   // complex()
                                   // construct complex(3) using
complex f = 3;
                                   // 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 ]

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]

<sup>3</sup> [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]

§ 12.6.1 254

[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]

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\mbox{-}initializer:
       : mem-initializer-list
mem\text{-}initializer\text{-}list:
       mem-initializer ... _{opt}
       mem-initializer, mem-initializer-list...ont
mem-initializer:
       mem-initializer-id ( expression-list_{opt})
       mem\text{-}initializer\text{-}id\ braced\text{-}init\text{-}list
mem-initializer-id:
       class-or-decltype
       identifier
```

- In a mem-initializer-id an initial unqualified identifier is looked up in the scope of the constructor's class and, if not found in that scope, it is looked up in the scope containing the constructor's definition. [Note: If the constructor's class contains a member with the same name as a direct or virtual base class of the class, a mem-initializer-id naming the member or base class and composed of a single identifier refers to the class member. A mem-initializer-id for the hidden base class may be specified using a qualified name. - end note | Unless the mem-initializer-id names the constructor's class, a non-static data member of the constructor's class, or a direct or virtual base of that class, the mem-initializer is ill-formed.
- A mem-initializer-list can initialize a base class using any class-or-decltype that denotes that base class type. [Example:

```
struct A { A(); };
   typedef A global_A;
   struct B { };
   struct C: public A, public B { C(); };
   C::C(): global_A() { }
                                  // mem-initializer for base A
— end example]
```

If a mem-initializer-id is ambiguous because it designates both a direct non-virtual base class and an inherited virtual base class, the mem-initializer is ill-formed. [Example:

```
struct A { A(); };
struct B: public virtual A { };
struct C: public A, public B { C(); };
C::C(): A() \{ \}
                                  // ill-formed: which A?
```

- end example]
- <sup>5</sup> A ctor-initializer may initialize a variant member of the constructor's class. If a ctor-initializer specifies more than one mem-initializer for the same member or for the same base class, the ctor-initializer is ill-formed.
- A mem-initializer-list can delegate to another constructor of the constructor's class using any class-ordecltype that denotes the constructor's class itself. If a mem-initializer-id designates the constructor's class, it shall be the only mem-initializer; the constructor is a delegating constructor, and the constructor selected by the mem-initializer is the target constructor. The principal constructor is the first constructor invoked in the construction of an object (that is, not a target constructor for that object's construction). The target constructor is selected by overload resolution. Once the target constructor returns, the body of the delegating constructor is executed. If a constructor delegates to itself directly or indirectly, the program is ill-formed; no diagnostic is required. [Example:

```
struct C {
```

§ 12.6.2 255

The expression-list or braced-init-list in a mem-initializer is used to initialize the designated subobject (or, in the case of a delegating constructor, the complete class object) according to the initialization rules of 8.5 for direct-initialization.

[Example:

— end example]

```
struct B1 { B1(int); /* ... */ };
struct B2 { B2(int); /* ... */ };
struct D : B1, B2 {
  D(int);
  B1 b;
  const int c;
};
D::D(int a) : B2(a+1), B1(a+2), c(a+3), b(a+4)
  { /* ... */ }
  D d(10);
```

- end example] The initialization performed by each mem-initializer constitutes a full-expression. Any expression in a mem-initializer is evaluated as part of the full-expression that performs the initialization. A mem-initializer where the mem-initializer-id denotes a virtual base class is ignored during execution of a constructor of any class that is not the most derived class.
- In a non-delegating constructor, 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).

[Note: An abstract class (10.4) is never a most derived class, thus its constructors never initialize virtual base classes, therefore the corresponding mem-initializers may be omitted. —  $end\ note$ ] An attempt to initialize more than one non-static data member of a union renders the program ill-formed. After the call to a constructor for class X has completed, if a member of X is neither initialized nor given a value during execution of the compound-statement of the body of the constructor, the member has indeterminate value. [Example:

§ 12.6.2

```
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
};
```

<sup>9</sup> 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

```
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]

- <sup>10</sup> In a non-delegating constructor, 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 mem-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 mem-initializers).
  - Finally, the *compound-statement* of the constructor body is executed.

[Note: The declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization. — end note]

11 [Example:

— end example]

```
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) { /* ... */ }
```

§ 12.6.2

— end example]

Names in the expression-list or braced-init-list of a mem-initializer are evaluated in the scope of the constructor for which the mem-initializer is specified. [Example:

```
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();
                      // undefined: calls member function
  B() : A(f()),
                      // but base A not yet initialized
  j(f()) { }
                      // well-defined: bases are all initialized
};
class C {
public:
  C(int);
class D : public B, C {
  int i;
public:
  D() : C(f()),
                      // undefined: calls member function
```

§ 12.6.2 258

```
i(f()) { }  // but base C not yet initialized
  i(f()) { }  // well-defined: bases are all initialized
};
--end example
```

[Note: 12.7 describes the result of virtual function calls, typeid and dynamic\_casts during construction for the well-defined cases; that is, describes the polymorphic behavior of an object under construction. — end note]

A mem-initializer followed by an ellipsis is a pack expansion (14.5.3) that initializes the base classes specified by a pack expansion in the base-specifier-list for the class. [Example:

```
template<class... Mixins>
class X : public Mixins... {
public:
    X(const Mixins&... mixins) : Mixins(mixins)... { }
};
    —end example]
```

#### 12.7 Construction and destruction

[class.cdtor]

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:

```
struct X { int i; };
                                                 // non-trivial
     struct Y : X { Y(); };
     struct A { int a; };
     struct B : public A { int j; Y y; };
                                                 // non-trivial
     extern B bobj;
                                                 // OK
     B* pb = \&bobj;
                                                 // undefined, refers to base class member
     int* p1 = &bobj.a;
     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;
                                                 //OK, X is a trivial class
     int* p3 = &xobj.i;
     X xobj;
<sup>2</sup> For another example,
     struct W { int j; };
     struct X : public virtual W { };
     struct Y {
       int *p;
       Y(): p(&x.j) { // undefined, x is not yet constructed
     };
```

- end example]

To explicitly or implicitly convert a pointer (a glvalue) referring to an object of class X to a pointer (reference) to a direct or indirect base class B of X, the construction of X and the construction of all of its direct or indirect bases that directly or indirectly derive from B shall have started and the destruction of these classes

§ 12.7 259

shall not have completed, otherwise the conversion results in undefined behavior. To form a pointer to (or access the value of) a direct non-static member of an object obj, the construction of obj shall have started and its destruction shall not have completed, otherwise the computation of the pointer value (or accessing the member value) results in undefined behavior. [Example:

```
struct A { };
struct B : virtual A { };
struct C : B { };
struct D : virtual A { D(A*); };
struct X { X(A*); };
struct E : C, D, X {
  E() : D(this),
                        // undefined: upcast from E* to A*
                        // might use path E* \rightarrow D* \rightarrow A*
                        // but D is not constructed
                        // D((C*)this), // defined:
                        // E* \rightarrow C* defined because E() has started
                        // and C* \rightarrow A* defined because
                        // C fully constructed
  X(this) {
                        // defined: upon construction of X,
                        // C/B/D/A sublattice is fully constructed
  }
};
```

— end example]

Member functions, including virtual functions (10.3), can be called during construction or destruction (12.6.2). When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, and the object to which the call applies is the object (call it x) under construction or destruction, the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access (5.2.5) and the object expression refers to the complete object of x or one of that object's base class subobjects but not x or one of its base class subobjects, the behavior is undefined. [Example:

```
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) {
                    // calls V::f, not A::f
  f();
```

§ 12.7 260

— end example]

- The typeid operator (5.2.8) can be used during construction or destruction (12.6.2). When typeid is used in a constructor (including 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.
- dynamic\_casts (5.2.7) can be used during construction or destruction (12.6.2). When a dynamic\_cast is used in a constructor (including the mem-initializer or brace-or-equal-initializer for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of the dynamic\_cast refers to the object under construction or destructor's class. If the operand of the dynamic\_cast refers to the object under construction or destructor and the static type of the operand is not a pointer to or object of the constructor or destructor's own class or one of its bases, the dynamic\_cast results in undefined behavior.

```
[Example:
 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) {
                                     // type_info for B
   typeid(*this);
                                     // well-defined: *v has type V, a base of B
   typeid(*v);
                                     // yields type_info for B
   typeid(*a);
                                     // undefined behavior: type A not a base of B
   dynamic_cast<B*>(v);
                                     // well-defined: v of type V*, V base of B
                                     // results in B*
   dynamic_cast<B*>(a);
                                     // undefined behavior,
                                     // a has type A*, A not a base of B
 }
— end example]
```

# 12.8 Copying and moving class objects

[class.copy]

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&, const X&, volatile X& or const volatile X&, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [Example: X::X(const X&) and X::X(X&,int=1) are copy constructors.

3 A non-template constructor for class X is a move constructor if its first parameter is of type X&&, const X&&, volatile X&&, or const volatile X&&, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [Example: Y::Y(Y&&) is a move constructor.

— end example]

4 [Note: All forms of copy/move constructor may be declared for a class. [Example:

 $-end \ example ] -end \ note ]$ 

<sup>5</sup> [Note: If a class X only has a copy constructor with a parameter of type X&, an initializer of type const X or volatile X cannot initialize an object of type (possibly cv-qualified) X. [Example:

A declaration of a constructor for a class X is ill-formed if its first parameter is of type (optionally cv-qualified) X and either there are no other parameters or else all other parameters have default arguments. A member function template is never instantiated to produce such a constructor signature. [Example:

```
struct S {
  template<typename T> S(T);
  S();
};
```

— end example]

If the class definition does not explicitly declare a copy constructor, one is declared *implicitly*. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted (8.4). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. Thus, for the class definition

```
struct X {
   X(const X&, int);
};
a copy constructor is implicitly-declared. If the user-declared constructor is later defined as
X::X(const X& x, int i =0) { /* ... */ }
```

then any use of X's copy constructor is ill-formed because of the ambiguity; no diagnostic is required.

<sup>8</sup> The implicitly-declared copy constructor for a class X will have the form

```
X::X(const X&)
```

if

- each direct or virtual base class B of X has a copy constructor whose first parameter is of type const B& or const volatile B&, and
- for all the non-static data members of X that are of a class type M (or array thereof), each such class type has a copy constructor whose first parameter is of type const M& or const volatile M&. 119

Otherwise, the implicitly-declared copy constructor will have the form

```
X::X(X\&)
```

- $^{9}$  If the definition of a class X does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if
  - X does not have a user-declared copy constructor,
  - X does not have a user-declared copy assignment operator,
  - X does not have a user-declared move assignment operator,
  - X does not have a user-declared destructor, and
  - the move constructor would not be implicitly defined as deleted.

[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:

<sup>119)</sup> This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a volatile lvalue; see C.1.9.

- 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,
- 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,
- any direct or virtual base class or non-static data member of a type with a destructor that is deleted or inaccessible from the defaulted constructor,
- for the copy constructor, a non-static data member of rvalue reference type, 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.
- <sup>12</sup> A copy/move constructor for class X is trivial if it is not user-provided and if
  - class X has no virtual functions (10.3) and no virtual base classes (10.1), and
  - 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 odr-used (3.2) 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<sup>120</sup> or when it is explicitly defaulted after its first declaration. [Note: The copy/move constructor is implicitly defined even if the implementation elided its odr-use (3.2, 12.2). end note] If the implicitly-defined constructor would satisfy the requirements of a constexpr constructor (7.1.5), the implicitly-defined constructor is constexpr.
- 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). end note]
- The implicitly-defined copy/move constructor for a non-union class X performs a memberwise copy/move of its bases and members. [Note: brace-or-equal-initializers of non-static data members are ignored. See also the example in 12.6.2. end note] The order of initialization is the same as the order of initialization of bases and members in a user-defined constructor (see 12.6.2). Let x be either the parameter of the constructor or, for the move constructor, an xvalue referring to the parameter. Each base or non-static data member is copied/moved in the manner appropriate to its type:
  - if the member is an array, each element is direct-initialized with the corresponding subobject of x;
  - if a member m has rvalue reference type T&&, it is direct-initialized with static\_cast<T&&>(x.m);
  - otherwise, the base or member is direct-initialized with the corresponding base or member of x.

<sup>120)</sup> See 8.5 for more details on direct and copy initialization.

Virtual base class subobjects shall be initialized only once by the implicitly-defined copy/move constructor (see 12.6.2).

- 16 The implicitly-defined copy/move constructor for a union X copies the object representation (3.9) of X.
- A user-declared *copy* assignment operator X::operator= is a non-static non-template member function of class X with exactly one parameter of type X, X&, const X&, volatile X& or const volatile X&. 121 [Note: An overloaded assignment operator must be declared to have only one parameter; see 13.5.3. end note] [Note: More than one form of copy assignment operator may be declared for a class. end note] [Note: If a class X only has a copy assignment operator with a parameter of type X&, an expression of type const X cannot be assigned to an object of type X. [Example:

- end example ] - end note ]

If the class definition does not explicitly declare a copy assignment operator, one is declared *implicitly*. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy assignment operator is defined as deleted; otherwise, it is defined as defaulted (8.4). The latter case is deprecated if the class has a user-declared copy constructor or a user-declared destructor. The implicitly-declared copy assignment operator for a class X will have the form

```
X& X::operator=(const X&)
if
```

- each direct base class B of X has a copy assignment operator whose parameter is of type const B&, const volatile B& or B, and
- for all the non-static data members of X that are of a class type M (or array thereof), each such class type has a copy assignment operator whose parameter is of type const M&, const volatile M& or M. 122

Otherwise, the implicitly-declared copy assignment operator will have the form

```
X& X::operator=(X&)
```

- A user-declared move assignment operator X::operator= is a non-static non-template member function of class X with exactly one parameter of type X&&, const X&&, volatile X&&, or const volatile X&&. [Note: An overloaded assignment operator must be declared to have only one parameter; see 13.5.3. end note]

  [Note: More than one form of move assignment operator may be declared for a class. end note]
- If the definition of a class X does not explicitly declare a move assignment operator, one will be implicitly declared as defaulted if and only if
  - X does not have a user-declared copy constructor,

<sup>121)</sup> 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.

<sup>122)</sup> This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a volatile lvalue; see C.1.9.

- X does not have a user-declared move constructor,
- X does not have a user-declared copy assignment operator,
- X does not have a user-declared destructor, 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& X::operator=(X&&);
```

- The implicitly-declared copy/move assignment operator for class X has the return type X&; it returns the object for which the assignment operator is invoked, that is, the object assigned to. An implicitly-declared copy/move assignment operator is an inline public member of its class.
- 23 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 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.
- <sup>25</sup> A copy/move assignment operator for class X is trivial if it is not user-provided and if

- class X has no virtual functions (10.3) and no virtual base classes (10.1), and
- 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 it is odr-used (3.2) (e.g., when it is selected by overload resolution to assign to an object of its class type) or when it is explicitly defaulted after its first declaration.
- 27 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). end note]
- The implicitly-defined copy/move assignment operator for a non-union class X performs memberwise copy/move assignment of its subobjects. The direct base classes of X are assigned first, in the order of their declaration in the base-specifier-list, and then the immediate non-static data members of X are assigned, in the order in which they were declared in the class definition. Let x be either the parameter of the function or, for the move operator, an xvalue referring to the parameter. Each subobject is assigned in the manner appropriate to its type:
  - if the subobject is of class type, as if by a call to **operator=** with the subobject as the object expression and the corresponding subobject of **x** as a single function argument (as if by explicit qualification; that is, ignoring any possible virtual overriding functions in more derived classes);
  - if the subobject is an array, each element is assigned, in the manner appropriate to the element type;
  - if the subobject is of scalar type, the built-in assignment operator is used.

It is unspecified whether subobjects representing virtual base classes are assigned more than once by the implicitly-defined copy assignment operator. [Example:

```
struct V { };
struct A : virtual V { };
struct B : virtual V { };
struct C : B, A { };
```

It is unspecified whether the virtual base class subobject V is assigned twice by the implicitly-defined copy assignment operator for C. —  $end\ example$  [ Note: This does not apply to move assignment, as a defaulted move assignment operator is deleted if the class has virtual bases. —  $end\ note$  ]

- The implicitly-defined copy assignment operator for a union X copies the object representation (3.9) of X.
- A program is ill-formed if the copy/move constructor or the copy/move assignment operator for an object is implicitly odr-used and the special member function is not accessible (Clause 11). [Note: Copying/moving one object into another using the copy/move constructor or the copy/move assignment operator does not change the layout or size of either object. end note]
- 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. This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

<sup>123)</sup> 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.

— in a return statement in a function with a class return type, when the expression is the name of a non-volatile automatic object (other than a function or catch-clause parameter) with the same cv-unqualified type as the function return type, the copy/move operation can be omitted by constructing the automatic object directly into the function's return value

- in a throw-expression, when the operand is the name of a non-volatile automatic object (other than a function or catch-clause parameter) whose scope does not extend beyond the end of the innermost enclosing try-block (if there is one), the copy/move operation from the operand to the exception object (15.1) can be omitted by constructing the automatic object directly into the exception object
- 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:
class Thing {
public:
   Thing();
   ~Thing();
   Thing(const Thing&);
};

Thing f() {
   Thing t;
   return t;
}
Thing t2 = f();
```

Here the criteria for elision can be combined to eliminate two calls to the copy constructor of class Thing: the copying of the local automatic object t into the temporary object for the return value of function f() and the copying of that temporary object into object t2. Effectively, the construction of the local object t can be viewed as directly initializing the global object t2, and that object's destruction will occur at program exit. Adding a move constructor to Thing has the same effect, but it is the move construction from the temporary object to t2 that is elided. — end example]

When the criteria for elision of a copy operation are met or would be met save for the fact that the source object is a function parameter, and the object to be copied is designated by an Ivalue, 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 Ivalue. [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]

class Thing {
public:
 Thing();

[Example:

```
~Thing();
   Thing(Thing&&);
 private:
   Thing(const Thing&);
 };
 Thing f(bool b) {
   Thing t;
   if (b)
                                    // OK: Thing(Thing&&) used (or elided) to throw t
      throw t;
                                    // OK: Thing(Thing&&) used (or elided) to return t
   return t;
 }
 Thing t2 = f(false);
                                    // OK: Thing(Thing&&) used (or elided) to construct t2
— end example]
```

# 12.9 Inheriting constructors

[class.inhctor]

- A using-declaration (7.3.3) that names a constructor implicitly declares a set of inheriting constructors. The candidate set of inherited constructors from the class X named in the using-declaration consists of actual constructors and notional constructors that result from the transformation of defaulted parameters as follows:
  - 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.
- <sup>2</sup> 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).
- <sup>3</sup> 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]
- <sup>4</sup> 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).
- <sup>5</sup> [Note: Default and copy/move constructors may be implicitly declared as specified in 12.1 and 12.8. end note]

6 [Example:

```
struct B1 {
  B1(int);
};
struct B2 {
 B2(int = 13, int = 42);
struct D1 : B1 {
  using B1::B1;
struct D2 : B2 {
  using B2::B2;
};
 The candidate set of inherited constructors in D1 for B1 is
— B1(const B1&)
— B1(B1&&)
— B1(int)
 The set of constructors present in D1 is
— D1(), implicitly-declared default constructor, ill-formed if odr-used
— 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 \tt 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]
- <sup>7</sup> [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:

- An inheriting constructor for a class is implicitly defined when it is odr-used (3.2) to create an object of its class type (1.8). An implicitly-defined inheriting constructor performs the set of initializations of the class that would be performed by a user-written inline constructor for that class with a mem-initializer-list whose only mem-initializer has a mem-initializer-id that names the base class denoted in the nested-name-specifier of the using-declaration and an expression-list as specified below, and where the compound-statement in its function body is empty (12.6.2). If that user-written constructor would be ill-formed, the program is ill-formed. Each expression in the expression-list is of the form static\_cast<T&&>(p), where p is the name of the corresponding constructor parameter and T is the declared type of p.
- <sup>9</sup> [Example:

 $-end \ example$ ]  $-end \ note$ ]

```
struct B1 {
  B1(int) { }
};
struct B2 {
  B2(double) { }
};
struct D1 : B1 {
  using B1::B1;
                      // implicitly declares D1(int)
  int x;
};
void test() {
                      // OK: d.x is not initialized
  D1 d(6);
  D1 e;
                      // error: D1 has no default constructor
struct D2 : B2 {
                      // OK: implicitly declares D2(double)
  using B2::B2;
```

```
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; }
};</pre>
```

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

[over]

When two or more different declarations are specified for a single name in the same scope, that name is said to be *overloaded*. By extension, two declarations in the same scope that declare the same name but with different types are called *overloaded declarations*. Only function and function template declarations can be overloaded; variable and type declarations cannot be overloaded.

When an overloaded function name is used in a call, which overloaded function declaration is being referenced is determined by comparing the types of the arguments at the point of use with the types of the parameters in the overloaded declarations that are visible at the point of use. This function selection process is called overload resolution and is defined in 13.3. [Example:

```
double abs(double);
int abs(int);

abs(1);  // calls abs(int);
abs(1.0);  // calls abs(double);

— end example]
```

#### 13.1 Overloadable declarations

[over.load]

- 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]
- <sup>2</sup> Certain function declarations cannot be overloaded:
  - Function declarations that differ only in the return type cannot be overloaded.
  - Member function declarations with the same name and the same parameter-type-list cannot be overloaded if any of them is a static member function declaration (9.4). Likewise, member function template declarations with the same name, the same parameter-type-list, and the same template parameter lists cannot be overloaded if any of them is a static member function template declaration. The types of the implicit object parameters constructed for the member functions for the purpose of overload resolution (13.3.1) are not considered when comparing parameter-type-lists for enforcement of this rule. In contrast, if there is no static member function declaration among a set of member function declarations with the same name and the same parameter-type-list, then these member function declarations can be overloaded if they differ in the type of their implicit object parameter. [Example: the following illustrates this distinction:

§ 13.1 273

— 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:

- <sup>3</sup> [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) { /* ... */ }

-- 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:

§ 13.1 274

— Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent. That is, the function type is adjusted to become a pointer to function type (8.3.5). [Example:

```
void h(int());
void h(int (*)());
void h(int x()) { }

// redeclaration of h(int())

// definition of h(int())

void h(int (*x)()) { }

// ill-formed: redefinition of h(int())

-- end example]
```

— Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. [Example:

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. <sup>124</sup> In particular, for any type T, "pointer to T," "pointer to const T," and "pointer to volatile T" are considered distinct parameter types, as are "reference to T," "reference to const T," and "reference to volatile T."

— Two parameter declarations that differ only in their default arguments are equivalent. [Example: consider the following:

```
void f (int i, int j);
 void f (int i, int j = 99);
                                   // OK: redeclaration of f(int, int)
                                   // OK: redeclaration of f(int, int)
 void f (int i = 88, int j);
 void f ();
                                   // OK: overloaded declaration of f
 void prog () {
                                   // OK: call f(int, int)
     f (1, 2);
                                   // OK: call f(int, int)
     f (1);
                                   // Error: f(int, int) or f()?
     f ();
 }
-end \ example] -end \ note]
```

#### 13.2 Declaration matching

[over.dcl]

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:

§ 13.2 275

<sup>124)</sup> When a parameter type includes a function type, such as in the case of a parameter type that is a pointer to function, the const and volatile type-specifiers at the outermost level of the parameter type specifications for the inner function type are also ignored.

```
struct B {
    int f(int);
  };
  struct D : B {
    int f(const char*);
   Here D::f(const char*) hides B::f(int) rather than overloading it.
  void h(D* pd) {
    pd->f(1);
                                    // error:
                                    // D::f(const char*) hides B::f(int)
    pd->B::f(1);
                                    // OK
    pd->f("Ben");
                                    // OK, calls D::f
 — end example]
A locally declared function is not in the same scope as a function in a containing scope. [Example:
  void f(const char*);
  void g() {
    extern void f(int);
    f("asdf");
                                    // error: f(int) hides f(const char*)
                                    // so there is no f(const char*) in this scope
  }
  void caller () {
    extern void callee(int, int);
                                    // hides callee(int, int)
      extern void callee(int);
       callee(88, 99);
                                    // error: only callee(int) in scope
    }
  }
 — end example]
Different versions of an overloaded member function can be given different access rules. [Example:
  class buffer {
  private:
       char* p;
      int size;
  protected:
      buffer(int s, char* store) { size = s; p = store; }
  public:
      buffer(int s) { p = new char[size = s]; }
  };
 — end example]
```

#### Overload resolution 13.3

[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

§ 13.3 276

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

#### [over.match.funcs]

- <sup>1</sup> 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.
- <sup>2</sup> 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.
- <sup>3</sup> 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.
- <sup>4</sup> For non-static member functions, the type of the implicit object parameter is
  - "Ivalue reference to cv X" for functions declared without a ref-qualifier or with the & ref-qualifier
  - "rvalue reference to cv X" for functions declared with the && ref-qualifier

§ 13.3.1

where X is the class of which the function is a member and cv is the cv-qualification on the member function declaration. [Example: for a const member function of class X, the extra parameter is assumed to have type "reference to const X". — end example] For conversion functions, the function is considered to be a member of the class of the implied object argument for the purpose of defining the type of the implicit object parameter. For non-conversion functions introduced by a using-declaration into a derived class, the function is considered to be a member of the derived class for the purpose of defining the type of the implicit object parameter. For static member functions, the implicit object parameter is considered to match any object (since if the function is selected, the object is discarded). [Note: No actual type is established for the implicit object parameter of a static member function, and no attempt will be made to determine a conversion sequence for that parameter (13.3.3). — end note]

- 5 During overload resolution, the implied object argument is indistinguishable from other arguments. The implicit object parameter, however, retains its identity since conversions on the corresponding argument shall obey these additional rules:
  - no temporary object can be introduced to hold the argument for the implicit object parameter; and
  - no user-defined conversions can be applied to achieve a type match with it.

For non-static member functions declared without a ref-qualifier, an additional rule applies:

- even if the implicit object parameter is not const-qualified, an rvalue can be bound to the parameter as long as in all other respects the argument can be converted to the type of the implicit object parameter. [Note: The fact that such an argument is an rvalue does not affect the ranking of implicit conversion sequences (13.3.3.2). end note]
- <sup>6</sup> 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:

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. 125 A given name can refer to one or more function templates and also to a set of overloaded non-template functions. In such a case, the candidate functions generated from each function template are combined with the set of non-template candidate functions.

# 13.3.1.1 Function call syntax

[over.match.call]

In a function call (5.2.2)

postfix-expression ( expression-list $_{opt}$ )

§ 13.3.1.1 278

<sup>125)</sup> The process of argument deduction fully determines the parameter types of the function template specializations, i.e., the parameters of function template specializations contain no template parameter types. Therefore the function template specializations can be treated as normal (non-template) functions for the remainder of overload resolution.

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]

#### 13.3.1.1.1 Call to named function

[over.call.func]

Of interest in 13.3.1.1.1 are only those function calls in which the *postfix-expression* ultimately contains a name that 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:

 $postfix-expression:\\postfix-expression. id-expression\\postfix-expression \rightarrow id-expression$ 

primary-expression

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

- In qualified function calls, the name to be resolved is an *id-expression* and is preceded by 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<sup>126</sup>. Under this assumption, the *id-expression* in the call is looked up as a member function of T following the rules for looking up names in classes (10.2). The function declarations found by that lookup constitute the set of candidate functions. The argument list is the *expression-list* in the call augmented by the addition of the left operand of the . operator in the normalized member function call as the implied object argument (13.3.1).
- In unqualified function calls, the name is not qualified by an -> or . operator and has the more general form of a primary-expression. The name is looked up in the context of the function call following the normal rules for name lookup in function calls (3.4). The function declarations found by that lookup constitute the set of candidate functions. Because of the rules for name lookup, the set of candidate functions consists (1) entirely of non-member functions or (2) entirely of member functions of some class T. In case (1), the argument list is the same as the expression-list in the call. In case (2), the argument list is the expression-list in the call augmented by the addition of an implied object argument as in a qualified function call. If the keyword this (9.3.2) is in scope and refers to class T, or a derived class of T, then the implied object argument is (\*this). If the keyword this is not in scope or refers to another class, then a contrived object of type T becomes the implied object argument 127. 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

[over.call.object]

- 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().
- <sup>2</sup> In addition, for each non-explicit conversion function declared in T of the form

operator  $conversion\mbox{-type-id}$  ()  $attribute\mbox{-specifier-seq}_{opt}$   $cv\mbox{-qualifier}$  ;

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

§ 13.3.1.1.2 279

<sup>126)</sup> Note that cv-qualifiers on the type of objects are significant in overload resolution for both glvalue and class prvalue objects.

<sup>127)</sup> An implied object argument must be contrived to correspond to the implicit object parameter attributed to member functions during overload resolution. It is not used in the call to the selected function. Since the member functions all have the same implicit object parameter, the contrived object will not be the cause to select or reject a function.

to 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 <sup>128</sup>.

- 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.
- 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:

#### 13.3.1.2 Operators in expressions

[over.match.oper]

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:

§ 13.3.1.2

<sup>128)</sup> 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.

}

— end example]

If either operand has a type that is a class or an enumeration, a user-defined operator function might be declared that implements this operator or a user-defined conversion can be necessary to convert the operand to a type that is appropriate for a built-in operator. In this case, overload resolution is used to determine which operator function or built-in operator is to be invoked to implement the operator. Therefore, the operator notation is first transformed to the equivalent function-call notation as summarized in Table 11 (where @ denotes one of the operators covered in the specified subclause).

Subclause	Expression	As member function	As non-member function
13.5.1	@a	(a).operator@()	operator@ (a)
13.5.2	a@b	(a).operator@ (b)	operator@ (a, b)
13.5.3	a=b	(a).operator= (b)	
13.5.5	a[b]	(a).operator[](b)	
13.5.6	a->	(a).operator-> ()	
13.5.7	a@	(a).operator@ (0)	operator@ (a, 0)

Table 11 — Relationship between operator and function call notation

- <sup>3</sup> 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 <code>operator@</code> in the context of the expression according to the usual rules for name lookup in unqualified function calls (3.4.2) except that all member functions are ignored. However, if no operand has a class type, only those non-member functions in the lookup set that have a first parameter of type T1 or "reference to (possibly cv-qualified) T1", when T1 is an enumeration type, or (if there is a right operand) a second parameter of type T2 or "reference to (possibly cv-qualified) T2", when T2 is an enumeration type, are candidate functions.
  - For the operator ,, the unary operator &, or the operator ->, the built-in candidates set is empty. For all other operators, the built-in candidates include all of the candidate operator functions defined in 13.6 that, compared to the given operator,
    - have the same operator name, and
    - accept the same number of operands, and
    - accept operand types to which the given operand or operands can be converted according to 13.3.3.1, and
    - do not have the same parameter-type-list as any non-template non-member candidate.
- <sup>4</sup> 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.

§ 13.3.1.2

- 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.<sup>129</sup> [Example:

```
struct A {
  operator int();
};
A operator+(const A&, const A&);
void m() {
  A a, b;
  a + b;  // operator+(a,b) chosen over int(a) + int(b)
}
```

— end example]

- <sup>7</sup> 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.
- <sup>8</sup> The second operand of operator -> is ignored in selecting an operator-> function, and is not an argument when the operator-> function is called. When operator-> returns, the operator -> is applied to the value returned, with the original second operand. <sup>130</sup>
- <sup>9</sup> If the operator is the operator ,, the unary operator &, or the operator ->, and there are no viable functions, then the operator is assumed to be the built-in operator and interpreted according to Clause 5.
- <sup>10</sup> [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:

#### 13.3.1.3 Initialization by constructor

[over.match.ctor]

When objects of class type are direct-initialized (8.5), or copy-initialized from an expression of the same or a derived class type (8.5), overload resolution selects the constructor. For direct-initialization, the candidate functions are all the constructors of the class of the object being initialized. For copy-initialization, the candidate functions are all the converting constructors (12.3.1) of that class. The argument list is the expression-list or assignment-expression of the initializer.

## 13.3.1.4 Copy-initialization of class by user-defined conversion

[over.match.copy]

Under the conditions specified in 8.5, as part of a copy-initialization of an object of class type, a user-defined conversion can be invoked to convert an initializer expression to the type of the object being initialized.

§ 13.3.1.4 282

<sup>129)</sup> If the set of candidate functions is empty, overload resolution is unsuccessful.

<sup>130)</sup> If the value returned by the operator-> function has class type, this may result in selecting and calling another operator-> function. The process repeats until an operator-> function returns a value of non-class type.

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 constructor that takes a reference to possibly cv-qualified T as its first argument, 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 lyalues or xyalues, depending on the type of reference, of type X and are therefore considered to yield X for this process of selecting candidate functions.
- In both cases, the argument list has one argument, which is the initializer expression. [Note: This argument will be compared against the first parameter of the constructors and against the implicit object parameter of the conversion functions. —end note]

#### 13.3.1.5 Initialization by conversion function

[over.match.conv]

- Under the conditions specified in 8.5, as part of an initialization of an object of nonclass type, a conversion function can be invoked to convert an initializer expression of class type to the type of the object being initialized. Overload resolution is used to select the conversion function to be invoked. Assuming that "cv1 T" is the type of the object being initialized, and "cv S" is the type of the initializer expression, with S a class type, the candidate functions are selected as follows:
  - The conversion functions of S and its base classes are considered. Those non-explicit conversion functions that are not hidden within S and yield type T or a type that can be converted to type T via a standard conversion sequence (13.3.3.1.1) are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within S and yield type T or a type that can be converted to type T with a qualification conversion (4.4) are also candidate functions. Conversion functions that return a cv-qualified type are considered to yield the cv-unqualified version of that type for this process of selecting candidate functions. Conversion functions that return "reference to cv2 X" return lyalues or xyalues, depending on the type of reference, of type "cv2 X" and are therefore considered to yield X for this process of selecting candidate functions.
- The argument list has one argument, which is the initializer expression. [Note: This argument will be compared against the implicit object parameter of the conversion functions. —end note]

#### 13.3.1.6 Initialization by conversion function for direct reference binding [over.match.ref]

- <sup>1</sup> 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.
- 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]

§ 13.3.1.6 283

#### 13.3.1.7 Initialization by list-initialization

[over.match.list]

When objects of non-aggregate class type T are list-initialized (8.5.4), overload resolution selects the constructor in two phases:

- Initially, the candidate functions are the initializer-list constructors (8.5.4) of the class T and the argument list consists of the initializer list as a single argument.
- If no viable initializer-list constructor is found, overload resolution is performed again, where the candidate functions are all the constructors of the class T and the argument list consists of the elements of the initializer list.

If the initializer list has no elements and T has a default constructor, the first phase is omitted. In copy-list-initialization, if an explicit constructor is chosen, the initialization is ill-formed. [Note: This differs from other situations (13.3.1.3, 13.3.1.4), where only converting constructors are considered for copy-initialization. This restriction only applies if this initialization is part of the final result of overload resolution. —  $end\ note$ ]

#### 13.3.2 Viable functions

[over.match.viable]

- <sup>1</sup> 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.
- <sup>2</sup> 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).<sup>131</sup> 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 Ivalue reference to non-const cannot be bound to an rvalue and that an rvalue reference cannot be bound to an Ivalue can affect the viability of the function (see 13.3.3.1.4).

# 13.3.3 Best viable function

[over.match.best]

- Define ICSi(F) as follows:
  - if F is a static member function, ICS1(F) is defined such that ICS1(F) is neither better nor worse than ICS1(G) for any function G, and, symmetrically, ICS1(G) is neither better nor worse than ICS1(F)<sup>132</sup>; otherwise,
  - let ICSi(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.

§ 13.3.3

<sup>131)</sup> According to 8.3.6, parameters following the (m+1)-st parameter must also have default arguments.

<sup>132)</sup> 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.

Given these definitions, a viable function F1 is defined to be a *better* function than another viable function F2 if for all arguments i, ICSi(F1) is not a worse conversion sequence than ICSi(F2), and then

- for some argument j, ICSj(F1) is a better conversion sequence than ICSj(F2), 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 F1 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 F2 to the destination type. [Example:

- F1 is a non-template function and F2 is a function template specialization, or, if not that,
- F1 and F2 are function template specializations, and the function template for F1 is more specialized than the template for F2 according to the partial ordering rules described in 14.5.6.2.
- <sup>2</sup> 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<sup>133</sup>.

[Example:

```
void Fcn(const int*, short);
void Fcn(int*, int);
int i;
short s = 0;
void f() {
                                          // is ambiguous because
  Fcn(&i, s);
                                          // &i \rightarrow int* is better than &i \rightarrow const int*
                                          // but s \rightarrow short is also better than s \rightarrow int
                                          // calls Fcn(int*, int), because
  Fcn(&i, 1L);
                                          // &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 	o int* is better than &i 	o const int*
                                          // and c \rightarrow int is better than c \rightarrow short
}
```

§ 13.3.3

<sup>133)</sup> 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 W that W did not face might be at least as good as W, W cannot be the best function because at some point in the tournament W encountered another function W such that W was not better than W. 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.

```
— end example]
```

If the best viable function resolves to a function for which multiple declarations were found, and if at least two of these declarations — or the declarations they refer to in the case of *using-declarations* — specify a default argument that made the function viable, the program is ill-formed. [Example:

#### — ena example]

# 13.3.3.1 Implicit conversion sequences

[over.best.ics]

- 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).
- <sup>2</sup> 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.
- 3 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.
- <sup>8</sup> 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).
- <sup>9</sup> 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<sup>134</sup>. 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.
- <sup>11</sup> The three forms of implicit conversion sequences mentioned above are defined in the following subclauses.

#### 13.3.3.1.1 Standard conversion sequences

[over.ics.scs]

- Table 12 summarizes the conversions defined in Clause 4 and partitions them into four disjoint categories: Lvalue Transformation, Qualification Adjustment, Promotion, and Conversion. [Note: These categories are orthogonal with respect to value category, cv-qualification, and data representation: the Lvalue Transformations do not change the cv-qualification or data representation of the type; the Qualification Adjustments do not change the value category or data representation of the type; and the Promotions and Conversions do not change the value category or cv-qualification of the type. —end note]
- [Note: As described in Clause 4, a standard conversion sequence is either the Identity conversion by itself (that is, no conversion) or consists of one to three conversions from the other four categories. At most one conversion from each category is allowed in a single standard conversion sequence. If there are two or more

134) 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,

```
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 \to C via constructor and // b \to A via constructor or conversion function.
```

If it were not for this rule, f(A) would be eliminated as a viable function for the call f(b) causing overload resolution to select f(C) as the function to call even though it is not clearly the best choice. On the other hand, if an f(B) were to be declared then f(b) would resolve to that f(B) because the exact match with f(B) is better than any of the sequences required to match f(A).

§ 13.3.3.1.1 287

conversions in the sequence, the conversions are applied in the canonical order: Lvalue Transformation, Promotion or Conversion, Qualification Adjustment. — end note]

Each conversion in Table 12 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.

Conversion	Category	Rank	Subclause
No conversions required	Identity	Exact Match	
Lvalue-to-rvalue conversion	Lvalue Transformation		4.1
Array-to-pointer conversion			4.2
Function-to-pointer conversion			4.3
Qualification conversions	Qualification Adjustment		4.4
Integral promotions	Promotion	Promotion	4.5
Floating point promotion			4.6
Integral conversions	Conversion	Conversion	4.7
Floating point conversions			4.8
Floating-integral conversions			4.9
Pointer conversions			4.10
Pointer to member conversions			4.11
Boolean conversions			4.12

Table 12 — Conversions

#### 13.3.3.1.2 User-defined conversion sequences

[over.ics.user]

- A user-defined conversion sequence consists of an initial standard conversion sequence followed by a user-defined conversion (12.3) followed by a second standard conversion sequence. If the user-defined conversion is specified by a constructor (12.3.1), the initial standard conversion sequence converts the source type to the type required by the argument of the constructor. If the user-defined conversion is specified by a conversion function (12.3.2), the initial standard conversion sequence converts the source type to the implicit object parameter of the conversion function.
- <sup>2</sup> 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).
- <sup>3</sup> 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.
- <sup>4</sup> 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

[over.ics.ellipsis]

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

[over.ics.ref]

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

§ 13.3.3.1.4 288

parameter type, in which case the implicit conversion sequence is a derived-to-base Conversion (13.3.3.1). [Example:

—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.
- <sup>3</sup> Except for an implicit object parameter, for which see 13.3.1, a standard conversion sequence cannot be formed if it requires binding an Ivalue reference other than a reference to a non-volatile const type to an rvalue or binding an rvalue reference to an Ivalue other than a function Ivalue. [Note: This means, for example, that a candidate function cannot be a viable function if it has a non-const Ivalue 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 Ivalue reference (see 8.5.3). end note]
- 4 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 "Ivalue reference to int" parameter can be a viable candidate even if the corresponding argument is an int bit-field. The formation of implicit conversion sequences treats the int bit-field as an int Ivalue and finds an exact match with the parameter. If the function is selected by overload resolution, the call will nonetheless be ill-formed because of the prohibition on binding a non-const Ivalue reference to a bit-field (8.5.3). end example]
- <sup>5</sup> 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 std::initializer\_list<X> or "array of X"<sup>135</sup> and all the elements of the initializer list can be implicitly converted to X, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to X. This conversion can be a user-defined conversion even in the context of a call to an initializer-list constructor. [Example:

135) Since there are no parameters of array type, this will only occur as the underlying type of a reference parameter.

§ 13.3.3.1.5

<sup>3</sup> Otherwise, if the parameter is a non-aggregate class X and overload resolution per 13.3.1.7 chooses a single best constructor of X to perform the initialization of an object of type X from the argument initializer list, the implicit conversion sequence is a user-defined conversion sequence. 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:

```
struct A {
   A(std::initializer_list<int>);
 void f(A);
 f( {'a', 'b'});
                             // OK: f(A(std::initializer_list<int>)) user-defined conversion
 struct B {
   B(int, double);
 };
 void g(B);
                             // OK: g(B(int,double)) user-defined conversion
 g( {'a', 'b'});
 g({1.0, 1,0});
                              // error: narrowing
 void f(B);
                              // error: ambiguous f(A) or f(B)
 f( {'a', 'b'});
 struct C {
   C(std::string);
 };
 void h(C);
                             // OK: h(C(std::string("foo")))
h({"foo"});
 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]
```

<sup>4</sup> 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:

```
struct A {
  int m1;
  double m2;
};
void f(A);
```

§ 13.3.3.1.5

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:

- 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:

— 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

[over.ics.rank]

- 1 13.3.3.2 defines a partial ordering of implicit conversion sequences based on the relationships better conversion sequence and better conversion. If an implicit conversion sequence S1 is defined by these rules to be a better conversion sequence than S2, then it is also the case that S2 is a worse conversion sequence than S1. If conversion sequence S1 is neither better than nor worse than conversion sequence S2, S1 and S2 are said to be indistinguishable conversion sequences.
- When comparing the basic forms of implicit conversion sequences (as defined in 13.3.3.1)
  - a standard conversion sequence (13.3.3.1.1) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence, and
  - a user-defined conversion sequence (13.3.3.1.2) is a better conversion sequence than an ellipsis conversion sequence (13.3.3.1.3).
- <sup>3</sup> Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules applies:

— Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence S2 if

- S1 is a proper subsequence of S2 (comparing the conversion sequences in the canonical form defined by 13.3.3.1.1, excluding any Lvalue Transformation; the identity conversion sequence is considered to be a subsequence of any non-identity conversion sequence) or, if not that,
- the rank of S1 is better than the rank of S2, or S1 and S2 have the same rank and are distinguishable by the rules in the paragraph below, or, if not that,
- S1 and S2 differ only in their qualification conversion and yield similar types T1 and T2 (4.4), respectively, and the cv-qualification signature of type T1 is a proper subset of the cv-qualification signature of type T2. [Example:

```
int f(const int *);
int f(int *);
int i;
int j = f(&i);  // calls f(int*)

— end example] or, if not that,
```

— S1 and S2 are reference bindings (8.5.3) and neither refers to an implicit object parameter of a non-static member function declared without a *ref-qualifier*, and S1 binds an rvalue reference to an rvalue and S2 binds an lvalue reference.

[Example:

```
int i;
int f1();
int&& f2();
int g(const int&);
int g(const int&&);
int j = g(i);
                                   // calls g(const int&)
int k = g(f1());
                                   // calls g(const int&&)
                                   // calls g(const int&&)
int 1 = g(f2());
struct A {
  A& operator<<(int);
  void p() &;
  void p() &&;
A& operator << (A&&, char);
A() << 1;
                                   // calls A::operator<<(int)</pre>
A() << 'c';
                                   // calls operator<<(A&&, char)
A a;
a << 1;
                                   // calls A::operator<<(int)
a << 'c';
                                   // calls A::operator<<(int)</pre>
                                   // calls A::p()&&
A().p();
a.p();
                                   // calls A::p()&
```

— S1 and S2 are reference bindings (8.5.3) and S1 binds an Ivalue reference to a function Ivalue and S2 binds an rvalue reference to a function Ivalue. [Example:

```
template<class T> int f(T&);
template<class T> int f(T&&);
void g();
int i1 = f(g);  // calls f(T&)
```

— end example or, if not that,

```
— end example]
```

— S1 and S2 are reference bindings (8.5.3), and the types to which the references refer are the same type except for top-level cv-qualifiers, and the type to which the reference initialized by S2 refers is more cv-qualified than the type to which the reference initialized by S1 refers. [Example:

```
int f(const int &);
 int f(int &);
 int g(const int &);
 int g(int);
 int i;
 int j = f(i);
                                     // calls f(int &)
 int k = g(i);
                                     // ambiguous
 struct X {
   void f() const;
   void f();
 };
 void g(const X& a, X b) {
                                     // calls X::f() const
// calls X::f()
   a.f();
   b.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 or aggregate initialization and the second standard conversion sequence of U1 is better than the second standard conversion sequence of U2. [Example:

- ena exampte]
- 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.
- <sup>4</sup> 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 *);
int f(B *);
int i = f(pc);  // calls f(B*)
```

- binding of an expression of type C to a reference of type B& is better than binding an expression of type C to a reference of type A&,
- conversion of A::\* to B::\* is better than conversion of A::\* to C::\*,
- conversion of C to B is better than conversion of C to A,
- conversion of B\* to A\* is better than conversion of C\* to A\*,
- binding of an expression of type B to a reference of type A& is better than binding an expression of type C to a reference of type A&,
- conversion of B::\* to C::\* is better than conversion of A::\* to C::\*, and
- conversion of B to A is better than conversion of C to A.

[Note: Compared conversion sequences will have different source types only in the context of comparing the second standard conversion sequence of an initialization by user-defined conversion (see 13.3.3); in all other contexts, the source types will be the same and the target types will be different. — end note]

#### 13.4 Address of overloaded function

[over.over]

- A use of an overloaded function name without arguments is resolved in certain contexts to a function, a pointer to function or a pointer to member function for a specific function from the overload set. A function template name is considered to name a set of overloaded functions in such contexts. The function selected is the one whose type is identical to the function type of the target type required in the context. [Note: That is, the class of which the function is a member is ignored when matching a pointer-to-member-function type. end note] The target can be
  - an object or reference being initialized (8.5, 8.5.3),
  - the left side of an assignment (5.17),
  - a parameter of a function (5.2.2),
  - a parameter of a user-defined operator (13.5),
  - the return value of a function, operator function, or conversion (6.6.3),
  - an explicit type conversion (5.2.3, 5.2.9, 5.4), or
  - a non-type template-parameter (14.3.2).

The overloaded function name can be preceded by the & operator. An overloaded function name shall not be used without arguments in contexts other than those listed. [Note: Any redundant set of parentheses surrounding the overloaded function name is ignored (5.1). — end note]

<sup>2</sup> 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

§ 13.4 294

specialization, which is added to the set of overloaded functions considered. [Note: As described in 14.8.1, if deduction fails and the function template name is followed by an explicit template argument list, the template-id is then examined to see whether it identifies a single function template specialization. If it does, the template-id is considered to be an Ivalue 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". 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.
- <sup>4</sup> If more than one function is selected, any function template specializations in the set are eliminated if the set also contains a non-template function, and any given function template specialization F1 is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of F1 according to the partial ordering rules of 14.5.6.2. After such eliminations, if any, there shall remain exactly one selected function.
- <sup>5</sup> [Example:

```
int f(double);
int f(int);
int (*pfd)(double) = &f;
                                   // selects f(double)
int (*pfi)(int) = &f;
                                   // selects f(int)
                                  // error: type mismatch
int (*pfe)(...) = &f;
int (&rfi)(int) = f;
                                  // selects f(int)
int (&rfd)(double) = f;
                                   // selects f(double)
void g() {
                                   // cast expression as selector
  (int (*)(int))&f;
}
```

The initialization of pfe is ill-formed because no f() with type int(...) has been declared, and not because of any ambiguity. For another example,

```
struct X {
  int f(int);
  static int f(long);
};
                                  // OK
int (X::*p1)(int) = \&X::f;
int
       (*p2)(int) = &X::f;
                                  // error: mismatch
       (*p3)(long) = &X::f;
                                  // OK
                                 // error: mismatch
int (X::*p4)(long) = \&X::f;
int (X::*p5)(int) = &(X::f);
                                 // error: wrong syntax for
                                 // pointer to member
       (*p6)(long) = &(X::f);
                                  // OK
int
```

— end example]

- [Note: If f() and g() are both overloaded functions, the cross product of possibilities must be considered to resolve f(&g), or the equivalent expression f(g). end note]
- Note: There are no standard conversions (Clause 4) of one pointer-to-function type into another. In particular, even if B is a public base of D, we have

§ 13.4 295

# 13.5 Overloaded operators

[over.oper]

<sup>1</sup> 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.

 $operator \hbox{-} function \hbox{-} id \hbox{:}$ 

[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.

<sup>3</sup> The following operators cannot be overloaded:

```
. .* :: ?:
```

nor can the preprocessing symbols # and ## (Clause 16).

Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement (13.5.1 - 13.5.7). They can be explicitly called, however, using the *operator-function-id* as the name of the function in the function call syntax (5.2.2). [Example:

— end example]

- The allocation and deallocation functions, operator new, operator new[], operator delete and operator delete[], are described completely in 3.7.4. The attributes and restrictions found in the rest of this subclause do not apply to them unless explicitly stated in 3.7.4.
- An operator function shall either be a non-static member function or be a non-member function 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 Ivalue when applied to basic types; this is not required by operator functions.
- <sup>8</sup> An operator function cannot have default arguments (8.3.6), except where explicitly stated below. Operator functions cannot have more or fewer parameters than the number required for the corresponding operator, as described in the rest of this subclause.
- Operators not mentioned explicitly in subclauses 13.5.3 through 13.5.7 act as ordinary unary and binary operators obeying the rules of 13.5.1 or 13.5.2.

#### 13.5.1 Unary operators

[over.unary]

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

§ 13.5.1 296

as either x.operator@() or operator@(x). If both forms of the operator function have been declared, the rules in 13.3.1.2 determine which, if any, interpretation is used. See 13.5.7 for an explanation of the postfix unary operators ++ and --.

<sup>2</sup> 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

[over.binary]

A binary operator shall be implemented either by a non-static member function (9.3) with one parameter or by a non-member function with two parameters. Thus, for any binary operator @, x@y can be interpreted as either x.operator@(y) or operator@(x,y). If both forms of the operator function have been declared, the rules in 13.3.1.2 determine which, if any, interpretation is used.

### 13.5.3 Assignment

[over.ass]

- An assignment operator shall be implemented by a non-static member function with exactly one parameter. Because a copy assignment operator operator= is implicitly declared for a class if not declared by the user (12.8), a base class assignment operator is always hidden by the copy assignment operator of the derived class.
- Any assignment operator, even the copy and move assignment operators, can be virtual. [Note: For a derived class D with a base class B for which a virtual copy/move assignment has been declared, the copy/move assignment operator in D does not override B's virtual copy/move assignment operator. [Example:

```
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)
                                  // ditto
  *bptr = 99;
                                  // calls D::operator=(const B&)
  bptr->operator=(dobj2);
                                  // ditto
  *bptr = dobj2;
                                  // calls implicitly-declared
  dobj1 = dobj2;
                                  // D::operator=(const D&)
}
```

# − end example ] − end note ]13.5.4 Function call

[over.call]

operator() shall be a non-static member function with an arbitrary number of parameters. It can have default arguments. It implements the function call syntax

```
postfix-expression ( expression-list_{opt})
```

where the *postfix-expression* evaluates to a class object and the possibly empty *expression-list* matches the parameter list of an operator() member function of the class. Thus, a call x(arg1,...) is interpreted as x.operator()(arg1,...) for a class object x of type T if T::operator()(T1, T2, T3) exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

# 13.5.5 Subscripting

[over.sub]

operator[] shall be a non-static member function with exactly one parameter. It implements the subscript-

§ 13.5.5

```
ing syntax

postfix-expression [ expression ]
```

postfix-expression [ braced-init-list ]

Thus, a subscripting expression x[y] is interpreted as x.operator[](y) for a class object x of type T if T::operator[](T1) exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3). [Example:

# 13.5.6 Class member access

[over.ref]

operator-> shall be a non-static member function taking no parameters. It implements the class member access syntax that uses ->.

```
postfix\text{-}expression \rightarrow \texttt{template}_{opt} \ id\text{-}expression \\ postfix\text{-}expression \rightarrow pseudo-destructor-name}
```

An expression  $x\to m$  is interpreted as  $(x.operator\to ())\to m$  for a class object x of type T if  $T::operator\to ()$  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

[over.inc]

The user-defined function called operator++ implements the prefix and postfix ++ operator. If this function is a member function with no parameters, or a non-member function with one parameter of class or enumeration type, it defines the prefix increment operator ++ for objects of that type. If the function is a member function with one parameter (which shall be of type int) or a non-member function with two parameters (the second of which shall be of type int), it defines the postfix increment operator ++ for objects of that type. When the postfix increment is called as a result of using the ++ operator, the int argument will have value zero. 

[Example:

```
struct X {
  X&
       operator++();
                                    // prefix ++a
  X
       operator++(int);
                                    // postfix a++
};
struct Y { };
     operator++(Y&);
                                    // prefix ++b
     operator++(Y&, int);
                                   // postfix b++
void f(X a, Y b) {
  ++a:
                                   // a.operator++();
  a++:
                                   // a.operator++(0);
                                   // operator++(b);
  ++b;
  b++;
                                    // operator++(b, 0);
                                   // explicit call: like ++a;
  a.operator++();
                                    // explicit call: like a++;
  a.operator++(0);
  operator++(b);
                                    // explicit call: like ++b;
  operator++(b, 0);
                                   // explicit call: like b++;
```

§ 13.5.7 298

<sup>136)</sup> Calling operator++ explicitly, as in expressions like a.operator++(2), has no special properties: The argument to operator++ is 2.

The prefix and postfix decrement operators -- are handled analogously.

# 13.5.8 User-defined literals

[over.literal]

- The *identifier* in a *literal-operator-id* is called a *literal suffix identifier*. [Note: some literal suffix identifiers are reserved for future standardization; see 17.6.4.3.5. end note]
- <sup>2</sup> A declaration whose declarator-id is a literal-operator-id shall be a declaration of a namespace-scope function or function template (it could be a friend function (11.3)), an explicit instantiation or specialization of a function template, or a using-declaration (7.3.3). A function declared with a literal-operator-id is a literal operator template.
- The declaration of a literal operator shall have a *parameter-declaration-clause* equivalent to one of the following:

```
const char*
unsigned long long int
long double
char
wchar_t
char16_t
char32_t
const char*, std::size_t
const wchar_t*, std::size_t
const char16_t*, std::size_t
const char32_t*, std::size_t
```

- 4 A raw literal operator is a literal operator with a single parameter whose type is const char\*.
- The declaration of a literal operator template shall have an empty parameter-declaration-clause and its template-parameter-list shall have a single template-parameter that is a non-type template parameter pack (14.5.3) with element type char.
- 6 Literal operators and literal operator templates shall not have C language linkage.
- 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]
- 8 [Example:

```
void operator "" _km(long double);
                                                         // OK
 string operator "" _i18n(const char*, std::size_t); // OK
                                                         // OK: UCN for lowercase pi
 template <char...> int operator "" \u03C0();
                                                         // error: ""E (with no intervening space)
 float operator ""E(const char*);
                                                         // is a single token
 float operator " " B(const char*);
                                                        // error: non-adjacent quotes
                                                        // error: invalid literal suffix identifier
 string operator "" 5X(const char*, std::size_t);
 double operator "" _miles(double);
                                                        // error: invalid parameter-declaration-clause
 template <char...> int operator "" j(const char*); // error: invalid parameter-declaration-clause
- end example]
```

### 13.6 Built-in operators

over.built

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

§ 13.6 299

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]

- <sup>2</sup> 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]
- <sup>3</sup> For every pair (T, VQ), where T is an arithmetic type, and VQ is either volatile or empty, there exist candidate operator functions of the form

```
VQ T& operator++(VQ T&);
T operator++(VQ T&, int);
```

<sup>4</sup> For every pair (T, VQ), where T is an arithmetic type other than bool, and VQ is either volatile or empty, there exist candidate operator functions of the form

```
VQ T& operator--(VQ T&);
T operator--(VQ T&, 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*VQ& operator++(T*VQ&);
T*VQ& operator--(T*VQ&);
T* operator++(T*VQ&, int);
T* operator--(T*VQ&, int);
```

<sup>6</sup> For every cy-qualified or cy-unqualified object type T, there exist candidate operator functions of the form

```
T& operator*(T*);
```

<sup>7</sup> For every function type T that does not have cv-qualifiers or a ref-qualifier, there exist candidate operator functions of the form

```
T& operator*(T*);
```

<sup>8</sup> For every type T there exist candidate operator functions of the form

```
T* operator+(T*);
```

 $^{9}$  For every promoted arithmetic type T, there exist candidate operator functions of the form

```
T operator+(T);
T operator-(T);
```

<sup>10</sup> For every promoted integral type T, there exist candidate operator functions of the form

```
T operator\sim(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

§ 13.6

CV12 T& 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
              operator *(L, R);
      LR
              operator/(L, R);
      LR
              operator+(L, R);
      LR
              operator-(L, R);
     bool
              operator<(L, R);
     bool
              operator>(L, R);
      bool
              operator <=(L, R);
      bool
              operator>=(L, R);
              operator==(L, R);
      bool
              operator!=(L, R);
      bool
       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*
              operator+(T*, std::ptrdiff_t);
      T\&
              operator[](T*, std::ptrdiff_t);
      T*
              operator-(T*, std::ptrdiff_t);
      T*
              operator+(std::ptrdiff_t, T*);
              operator[](std::ptrdiff_t, T*);
<sup>14</sup> For every T, where T is a pointer to object type, there exist candidate operator functions of the form
      std::ptrdiff_t
                       operator-(T , T );
   For every T, where T is an enumeration type, a pointer type, or std::nullptr t, there exist candidate
   operator functions of the form
      bool
              operator<(T , T );
      bool
              operator>(T , T );
     bool
              operator<=(T , T );
              operator>=(T, T);
      bool
      bool
              operator==(T, T);
              operator!=(T, T);
      bool
<sup>16</sup> For every pointer to member type T there exist candidate operator functions of the form
              operator==(T, T);
      bool
      bool
              operator!=(T, T);
  For every pair of promoted integral types L and R, there exist candidate operator functions of the form
      LR
               operator%(L, R);
      LR
               operator&(L , R );
      LR
               operator^(L , R );
      LR
               operator | (L , R );
      L
               operator<<(L , R );
               operator>>(L , R );
       where LR is the result of the usual arithmetic conversions between types L and R.
   For every triple (L, VQ, R), where L is an arithmetic type, VQ is either volatile or empty, and R is a
   promoted arithmetic type, there exist candidate operator functions of the form
      VQ L &
               operator=(VQ L &, R);
      VQ L &
               operator*=(VQ L &, R);
               operator/=(VQ L &, R);
      VQL &
      VQ L &
               operator+=(VQ L &, R);
      VQ L &
               operator = (VQ L &, R);
                                                                                                        301
   § 13.6
```

For every pair (T, VQ), where T is any type and VQ is either volatile or empty, there exist candidate operator functions of the form

```
T * VQ \& operator=(T * VQ \&, T *);
```

For every pair (T, VQ), where T is an enumeration or pointer to member type and VQ is either volatile or empty, there exist candidate operator functions of the form

```
VQ T & operator=(VQ T &, T);
```

For every pair (T, VQ), where T is a cv-qualified or cv-unqualified object type and VQ is either volatile or empty, there exist candidate operator functions of the form

```
T *VQ \& operator += (T *VQ \&, std::ptrdiff_t);

T *VQ \& operator -= (T *VQ \&, std::ptrdiff_t);
```

For every triple (L, VQ, R), where L is an integral type, VQ is either volatile or empty, and R is a promoted integral type, there exist candidate operator functions of the form

```
VQ L & operator%=(VQ L &, R );
VQ L & operator<<=(VQ L &, R );
VQ L & operator>>=(VQ L &, R );
VQ L & operator&=(VQ L &, R );
VQ L & operator^=(VQ L &, R );
VQ L & operator|=(VQ L &, R );
```

23 There also exist candidate operator functions of the form

```
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

```
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

```
T operator?:(bool, T , T );
```

§ 13.6 302

# 14 Templates

[temp]

A template defines a family of classes or functions or an alias for a family of types.

 $template ext{-}declaration:$ 

template < template-parameter-list > declaration

 $template ext{-}parameter ext{-}list:$ 

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, a member enumeration, 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.

- A template-declaration can appear only as a namespace scope or class scope declaration. In a function template declaration, the last component of the declarator-id shall not be a template-id. [Note: That last component may be an identifier, an operator-function-id, a conversion-function-id, or a literal-operator-id. In a class template declaration, if the class name is a simple-template-id, the declaration declares a class template partial specialization (14.5.5). end note]
- <sup>3</sup> In a template-declaration, explicit specialization, or explicit instantiation the *init-declarator-list* in the declaration shall contain at most one declarator. When such a declaration is used to declare a class template, no declarator is permitted.
- 4 A template name has linkage (3.5). A non-member function template can have internal linkage; any other template name shall have external linkage. Specializations (explicit or implicit) of a template that has internal linkage are distinct from all specializations in other translation units. A template, a template explicit specialization (14.7.3), and a class template partial specialization shall not have C linkage. Use of a linkage specification other than C or C++ with any of these constructs is conditionally-supported, with implementation-defined semantics. Template definitions shall obey the one definition rule (3.2). [Note: Default arguments for function templates and for member functions of class templates are considered definitions for the purpose of template instantiation (14.5) and must also obey the one definition rule. end note]
- A class template shall not have the same name as any other template, class, function, variable, enumeration, enumerator, namespace, or type in the same scope (3.3), except as specified in (14.5.5). Except that a function template can be overloaded either by (non-template) functions 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.
- <sup>6</sup> A function template, member function of a class template, or static data member of a class template shall be defined in every translation unit in which it is implicitly instantiated (14.7.1) unless the corresponding specialization is explicitly instantiated (14.7.2) in some translation unit; no diagnostic is required.

Templates 303

# 14.1 Template parameters

[temp.param]

The syntax for template-parameters is:

```
type-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 \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-list > class identifier_{opt} = id-expression \\ template < template-parameter-lis
```

[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 <sup>137</sup> 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,

```
template<class T> class myarray { /* ... */ };

template<class K, class V, template<class T> class C = myarray>
class Map {
    C<K> key;
    C<V> value;
};

— end note]
```

<sup>3</sup> 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,

Here, the template **f** has a *type-parameter* called **T**, rather than an unnamed non-type *template-parameter* of class **T**. — end note

- 4 A non-type template-parameter shall have one of the following (optionally cv-qualified) types:
  - integral or enumeration type,
  - pointer to object or pointer to function,
  - lvalue reference to object or lvalue reference to function,
  - pointer to member,

§ 14.1 304

<sup>137)</sup> 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.

```
— std::nullptr_t.
```

<sup>5</sup> [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.

<sup>6</sup> 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:

A non-type template-parameter shall not be declared to have floating point, class, or void type. [Example:

```
template<double d> class X; // error template<double* pd> class Y; // OK template<double& rd> class Z; // OK
```

— end example]

<sup>8</sup> 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:

- -end example
- 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.
- The set of default *template-arguments* available for use with a template declaration or definition is obtained by merging the default arguments from the definition (if in scope) and all declarations in scope in the same way default function arguments are (8.3.6). [Example:

```
template<class T1, class T2 = int> class A;
template<class T1 = int, class T2> class A;
is equivalent to
```

§ 14.1 305

```
template<class T1 = int, class T2 = int> class A;
— end example]
```

If a template-parameter of a class template or alias template has a default template-argument, each subsequent template-parameter shall either have a default template-argument supplied or be a template parameter pack. If a template-parameter of a primary class template or alias template is a template parameter pack, it shall be the last template-parameter. A template parameter pack of a function template shall not be followed by another template parameter unless that template parameter can be deduced or has a default argument (14.8.2). [Example:

```
template<class T1 = int, class T2> class B;  // error

// U cannot be deduced or specified
template<class... T, class... U> void f() { }
template<class... T, class U> void g() { }

— end example]
```

A template-parameter shall not be given default arguments by two different declarations in the same scope. [Example:

```
template<class T = int> class X; template<class T = int> class X { /*...*/ }; // error
```

- end example]

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:

— end example]

<sup>14</sup> 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:

- end example]

If a template-parameter is a type-parameter with an ellipsis prior to its optional identifier or is a parameter-declaration that declares a parameter pack (8.3.5), then the template-parameter is a template parameter pack (14.5.3). A template parameter pack that is a parameter-declaration whose type contains one or more unexpanded parameter packs is a pack expansion. Similarly, a template parameter pack that is a type-parameter with a template-parameter-list containing one or more unexpanded parameter packs is a pack expansion. A template parameter pack that is a pack expansion shall not expand a parameter pack declared in the same template-parameter-list. [Example:

§ 14.1 306

### 14.2 Names of template specializations

[temp.names]

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 \\ \end{cases}
```

[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]

- <sup>2</sup> For a *template-name* to be explicitly qualified by the template arguments, the name must be known to refer to a template.
- After name lookup (3.4) finds that a name is a template-name or that an operator-function-id 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 > 138 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:

138) 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.

§ 14.2 307

```
Y<X<(6>>1)>> x5; // OK
```

- end example]

When the name of a member template specialization appears after . or -> in a postfix-expression or after a nested-name-specifier in a qualified-id, and the object expression of the postfix-expression is type-dependent or the nested-name-specifier in the qualified-id refers to a dependent type, but the name is not a member of the current instantiation (14.6.2.1), the member template name must be prefixed by the keyword template. Otherwise the name is assumed to name a non-template. [Example:

— end example]

A name prefixed by the keyword template shall be a  $template \cdot id$  or the name shall refer to a class template. [Note: The keyword template may not be applied to non-template members of class templates. — end note] [Note: As is the case with the typename prefix, the template prefix is allowed in cases where it is not strictly necessary; i.e., when the nested-name-specifier or the expression on the left of the -> or . is not dependent on a template-parameter, or the use does not appear in the scope of a template. — end note] [Example:

```
template <class T> struct A {
   void f(int);
   template <class U> void f(U);
 };
 template <class T> void f(T t) {
   A<T> a;
                                      // OK: calls template
   a.template f<>(t);
   a.template f(t);
                                      // error: not a template-id
 template <class T> struct B {
   template <class T2> struct C { };
 // OK: T::template C names a class template:
 template <class T, template <class X> class TT = T::template C> struct D { };
 D<b<int> > db;
— end example]
```

- 6 A simple-template-id that names a class template specialization is a class-name (Clause 9).
- 7 A template-id that names an alias template specialization is a type-name.

#### 14.3 Template arguments

[temp.arg]

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:

§ 14.3

```
template<class T> class Array {
   T* v;
   int sz;
 public:
   explicit Array(int);
   T& operator[](int);
   T& elem(int i) { return v[i]; }
 };
 Array<int> v1(20);
 typedef std::complex<double> dcomplex; // std::complex is a standard
                                           // library template
 Array <dcomplex > v2(30);
 Array<dcomplex> v3(40);
 void bar() {
   v1[3] = 7;
   v2[3] = v3.elem(4) = dcomplex(7,8);
— end example]
```

— ena example]

<sup>2</sup> In a template-argument, an ambiguity between a type-id and an expression is resolved to a type-id, regardless of the form of the corresponding template-parameter. [Example:

— end example]

<sup>3</sup> 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:

— 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:

§ 14.3

<sup>139)</sup> 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.

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:

<sup>5</sup> 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:

- end example]
- <sup>6</sup> 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.
- <sup>7</sup> When the template in a *template-id* is an overloaded function template, both non-template functions in the overload set and function templates in the overload set for which the *template-arguments* do not match the *template-parameters* are ignored. If none of the function templates have matching *template-parameters*, the program is ill-formed.
- A template-argument followed by an ellipsis is a pack expansion (14.5.3).

# 14.3.1 Template type arguments

[temp.arg.type]

- <sup>1</sup> A template-argument for a template-parameter which is a type shall be a type-id.
- <sup>2</sup> [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
```

§ 14.3.1 310

```
X<A*> x2;  // OK
X<B> x3;  // OK
f(e1);  // OK
f(unnamed_obj);  // OK
f(b);  // OK
}
```

— end example [Note: A template type argument may be an incomplete type (3.9). — end note

<sup>3</sup> 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*:

# 14.3.2 Template non-type arguments

[temp.arg.nontype]

- A template-argument for a non-type, non-template template-parameter shall be one of:
  - for a non-type template-parameter of integral or enumeration type, a converted constant expression (5.19) of the type of the template-parameter; or
  - the name of a non-type template-parameter; or
  - a constant expression (5.19) that designates the address of an object with static storage duration and external or internal linkage or a function with external or internal linkage, including function templates and function template-ids but excluding non-static class members, expressed (ignoring parentheses) as & id-expression, except that the & may be omitted if the name refers to a function or array and shall be omitted 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.
- <sup>2</sup> [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:

```
template<class T, const char* p> class X {
    /* ... */
};

X<int, "Studebaker"> x1;  // error: string literal as template-argument

const char p[] = "Vivisectionist";

X<int,p> x2;  // OK

— end example] — end note]
```

<sup>3</sup> [Note: Addresses of array elements and names or addresses of non-static class members are not acceptable template-arguments. [Example:

§ 14.3.2 311

<sup>4</sup> [Note: Temporaries, unnamed lyalues, and named lyalues with no linkage are not acceptable template-arguments when the corresponding template-parameter has reference type. [Example:

- $-end \ example \ ] \ -end \ note \ ]$
- <sup>5</sup> 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, conversions permitted in a converted constant expression (5.19) 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 lyalue.
  - 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).

§ 14.3.2 312

— 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> xi;
                                    // array to pointer and qualification conversions
 struct Y { /* ... */ };
 template<const Y& b> struct Z { /* ... */ };
 Yy;
                                    // no conversion, but note extra cv-qualification
 Z < y > z;
 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 { /* ... */ };
                                    // selects f(int)
 A<&f> a;
— 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 an alias template, expressed as id-expression. When the template-argument names a class template, only primary class templates are considered when matching the template template argument with the corresponding parameter; partial specializations are not considered even if their parameter lists match that of the template template parameter.
- <sup>2</sup> Any partial specializations (14.5.5) associated with the primary class template are considered when a specialization based on the template template-parameter is instantiated. If a specialization is not visible at the point of instantiation, and it would have been selected had it been visible, the program is ill-formed; no diagnostic is required. [Example:

```
template<class T> class A {
                                   // primary template
   int x;
 template<class T> class A<T*> { // partial specialization
   long x;
 template<template<class U> class V> class C {
   V<int> y;
   V<int*> z;
 };
 C<A>c;
                                   // V<int> within C<A> uses the primary template,
                                   // so c.y.x has type int
                                    // V<int*> within C<A> uses the partial specialization,
                                   // so c.z.x has type long
- end example]
  [ Example:
```

§ 14.3.3

```
template<class T> class A { /* ... */ };
template < class T, class U = T > class B { /* ... */ };
template <class ... Types> class C { /* ... */ };
template<template<class> class P> class X { /* ... */ };
template<template<class ...> class Q> class Y { /* ... */ };
                     //OK
X<A> xa;
                     // ill-formed: default arguments for the parameters of a template argument are ignored
X<B> xb;
X<C>xc;
                     // ill-formed: a template parameter pack does not match a template parameter
                     // OK
Y<A> ya;
                     // OK
Y < B > yb;
                     // OK
Y<C> yc;
```

— end example]

A template-argument matches a template template-parameter (call it P) when each of the template parameters in the template-parameter-list of the template-argument's corresponding class template or alias template (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:

```
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;
                                   // OK: matches partial specialization of eval
 eval<A<int>> eA;
 eval<B<int, float>> eB;
                                   // OK: matches partial specialization of eval
                                   // error: C does not match TT in partial specialization
 eval<C<17>> eC;
 eval<D<int, 17>> eD;
                                   // error: D does not match TT in partial specialization
 eval<E<int, float>> eE;
                                   // error: E does not match TT in partial specialization
— end example]
```

# 14.4 Type equivalence

[temp.type]

- 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

§ 14.4 314

— 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:

```
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
template<class T, void(*err_fct)()> class list { /* ... */ };
list<int,&error_handler1> x1;
list<int,&error_handler2> x2;
list<int,&error_handler2> x3;
list<char,&error_handler2> x4;
 declares x2 and x3 to be of the same type. Their type differs from the types of x1 and x4.
template<class T> struct X { };
template<class> struct Y { };
template<class T> using Z = Y<T>;
X<Y<int>>y;
X<Z<int>>z;
 declares y and z to be of the same type. — end example ]
```

<sup>2</sup> If an expression *e* involves a template parameter, decltype(*e*) denotes a unique dependent type. Two such *decltype-specifiers* refer to the same type only if their *expressions* are equivalent (14.5.6.1). [*Note:* however, it may be aliased, e.g., by a *typedef-name*. — *end note*]

#### 14.5 Template declarations

[temp.decls]

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:

- end example [Note: However, this syntax is allowed in class template partial specializations (14.5.5). end note ]
- <sup>2</sup> 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.
- <sup>3</sup> Because an *alias-declaration* cannot declare a *template-id*, it is not possible to partially or explicitly specialize an alias template.

#### 14.5.1 Class templates

[temp.class]

A class template defines the layout and operations for an unbounded set of related types. [Example: a single class template List might provide a common definition for list of int, list of float, and list of pointers to Shapes. — end example]

[Example: An array class template might be declared like this:

```
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
```

§ 14.5.1 315

```
T& operator[](int);
T& elem(int i) { return v[i]; }
};
```

The prefix template <class T> specifies that a template is being declared and that a type-name T will be used in the declaration. In other words, Array is a parameterized type with T as its parameter. — end example |

When a member function, a member class, a member enumeration, a static data member or a member template of a class template is defined outside of the class template definition, the member definition is defined as a template definition in which the template-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:

```
template < class T1, class T2> struct A {
  void f1();
  void f2();
};
                                                          // OK
template<class T2, class T1> void A<T2,T1>::f1() { }
template<class T2, class T1> void A<T1,T2>::f2() { }
                                                          // error
template<class ... Types> struct B {
 void f3();
  void f4();
};
                                                           // OK
template<class ... Types> void B<Types ...>::f3() { }
template<class ... Types> void B<Types>::f4() { }
                                                           // error
  — end example]
```

<sup>4</sup> 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]

A member function of a class template may be defined outside of the class template definition in which it is declared. [Example:

```
template<class T> class Array {
   T* v;
   int sz;
public:
    explicit Array(int);
   T& operator[](int);
   T& elem(int i) { return v[i]; }
};

declares three function templates. The subscript function might be defined like this:
template<class T> T& Array<T>::operator[](int i) {
   if (i<0 || sz<=i) error("Array: range error");
   return v[i];
}

— end example]</pre>
```

§ 14.5.1.1 316

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.

### 14.5.1.2 Member classes of class templates

[temp.mem.class]

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,

#### 14.5.1.3 Static data members of class templates

[temp.static]

A definition for a static data member may be provided in a namespace scope enclosing the definition of the static member's class template. [Example:

```
template<class T> class X {
   static T s;
};
template<class T> T X<T>::s = 0;
```

 $-end \ example$ 

<sup>2</sup> An explicit specialization of a static data member declared as an array of unknown bound can have a different bound from its definition, if any. [Example:

```
template <class T> struct A {
   static int i[];
};
template <class T> int A<T>::i[4];  // 4 elements
template <> int A<int>::i[] = { 1 };  // OK: 1 element

— end example]
```

## 14.5.1.4 Enumeration members of class templates

[temp.mem.enum]

1 An enumeration member of a class template may be defined outside the class template definition. [Example:

```
template<class T> struct A {
   enum E : T;
};
A<int> a;
template<class T> enum A<T>::E : T { e1, e2 };
A<int>::E e = A<int>::e1;
```

§ 14.5.1.4 317

— end example]

## 14.5.2 Member templates

[temp.mem]

A template can be declared within a class or class template; such a template is called a member template. A member template can be defined within or outside its class definition or class template definition. A member template of a class template that is defined outside of its class template definition shall be specified with the template-parameters of the class template followed by the template-parameters of the member template. [Example:

<sup>2</sup> 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 templates. A normal (non-template) member function with a given name and type and a member function template of the same name, which could be used to generate a specialization of the same type, can both be declared in a class. When both exist, a use of that name and type refers to the non-template member unless an explicit template argument list is supplied. [Example:

```
template <class T> struct A {
    void f(int);
    template <class T2> void f(T2);
  };
  template <> void A<int>::f(int) { }
                                                             // non-template member
  template <> template <> void A<int>::f<>(int) { }
                                                             // template member
  int main() {
    A<char> ac;
    ac.f(1);
                       // non-template
    ac.f('c');
                       // template
    ac.f<>(1);
                       // template
 — end example]
A member function template shall not be virtual. [Example:
  template <class T> struct AA {
                                              // error
    template <class C> virtual void g(C);
                                              // OK
    virtual void f();
```

— end example]

};

<sup>4</sup> A specialization of a member function template does not override a virtual function from a base class. [Example:

```
class B {
  virtual void f(int);
};

class D : public B {
  template <class T> void f(T); // does not override B::f(int)
  void f(int i) { f<>(i); } // overriding function that calls
```

§ 14.5.2 318

```
// the template instantiation
```

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:

- $-end\ example$ ] [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.  $-end\ note$ ]
- <sup>6</sup> 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.
- 8 Overload resolution (13.3.3.2) and partial ordering (14.5.6.2) are used to select the best conversion function among multiple specializations of conversion function templates and/or non-template conversion functions.

#### 14.5.3 Variadic templates

**}**;

— end example]

[temp.variadic]

A template parameter pack is a template parameter that accepts zero or more template arguments. [Example:

<sup>2</sup> A function parameter pack is a function parameter that accepts zero or more function arguments. [Example:

- 3 A parameter pack is either a template parameter pack or a function parameter pack.
- <sup>4</sup> A pack expansion consists of a pattern and an ellipsis, the instantiation of which produces zero or more instantiations of the pattern in a list (described below). The form of the pattern depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:
  - In a function parameter pack (8.3.5); the pattern is the parameter-declaration without the ellipsis.

§ 14.5.3

- In a template parameter pack that is a pack expansion (14.1):
  - if the template parameter pack is a *parameter-declaration*; the pattern is the *parameter-declaration* without the ellipsis;
  - if the template parameter pack is a *type-parameter* with a *template-parameter-list*; the pattern is the corresponding *type-parameter* without the ellipsis.
- In an *initializer-list* (8.5); the pattern is an *initializer-clause*.
- In a base-specifier-list (Clause 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 an alignment-specifier (7.6.2); the pattern is the alignment-specifier without the ellipsis.
- In a capture-list (5.1.2); the pattern is a capture.
- In a sizeof... expression (5.3.3); the pattern is an *identifier*.

```
[Example:
```

— end example]

A parameter pack whose name appears within the pattern of a pack expansion is expanded by that pack expansion. An appearance of the name of a parameter pack is only expanded by the innermost enclosing pack expansion. The pattern of a pack expansion shall name one or more parameter packs that are not expanded by a nested pack expansion; such parameter packs are called *unexpanded* parameter packs in the pattern. All of the parameter packs expanded by a pack expansion shall have the same number of arguments specified. An appearance of a name of a parameter pack that is not expanded is ill-formed. [Example:

```
template<typename...> struct Tuple {};
template<tlass ... Args1> struct zip {
  template<class ... Args2> struct with {
    typedef Tuple<Pair<Args1, Args2> ... > type;
  };
};

typedef zip<short, int>::with<unsigned short, unsigned>::type T1;
    // T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned>>
typedef zip<short>::with<unsigned short, unsigned>::type T2;
    // error: different number of arguments specified for Args1 and Args2

template<class ... Args>
  void g(Args ... args) {
    // OK: Args is expanded by the function parameter pack args
    f(const_cast<const Args*>(&args)...); // OK: "Args" and "args" are expanded
```

§ 14.5.3

```
f(5 ...); // error: pattern does not contain any parameter packs
f(args); // error: parameter pack "args" is not expanded
f(h(args ...) + args ...); // OK: first "args" expanded within h, second
// "args" expanded within f
}

— end example]
```

The instantiation of a pack expansion that is not a sizeof... expression produces a list  $E_1, E_2, ..., E_N$ , where N is the number of elements in the pack expansion parameters. Each  $E_i$  is generated by instantiating the pattern and replacing each pack expansion parameter with its ith 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] When N is zero, the instantiation of the expansion produces an empty list. Such an instantiation does not alter the syntactic interpretation of the enclosing construct, even in cases where omitting the list entirely would otherwise be ill-formed or would result in an ambiguity in the grammar. [Example:

```
template<class... T> struct X : T... { };
template<class... T> void f(T... values) {
   X<T...> x(values...);
}

template void f<>(); // OK: X<> has no base classes
   // x is a variable of type X<> that is value-initialized
```

- end example]

<sup>7</sup> The instantiation of a sizeof... expression (5.3.3) produces an integral constant containing the number of elements in the parameter pack it expands.

14.5.4 Friends [temp.friend]

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 function template is found in the specified class or namespace, the friend declaration refers to the deduced specialization of that function template (14.8.2.6), otherwise,
- the name shall be an unqualified-id that declares (or redeclares) an ordinary (non-template) function.

[Example:

```
template<class T> class task;
template<class T> task<T>* preempt(task<T>*);

template<class T> class task {
  friend void next_time();
  friend void process(task<T>*);
  friend task<T>* preempt<T>(task<T>*);
  template<class C> friend int func(C);

friend class task<int>;
```

§ 14.5.4 321

```
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]

<sup>2</sup> 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:

— end example]

<sup>3</sup> 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:

```
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]

- <sup>4</sup> When a function is defined in a friend function declaration in a class template, the function is instantiated when the function is odr-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:

```
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();
   }
}
```

§ 14.5.4 322

```
};
  };
   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]
[Note: A friend declaration may first declare a member of an enclosing namespace scope (14.6.5). -end
A friend template shall not be declared in a local class.
Friend declarations shall not declare partial specializations. [Example:
   template<class T> class A { };
   class X {
```

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

template<class T> friend class A<T\*>; // error

[temp.class.spec]

- 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.
- <sup>2</sup> 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:

- end 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 declarations declare partial specializations of the primary template.  $-end\ example$ 

The template parameters are specified in the angle bracket enclosed list that immediately follows the keyword template. For partial specializations, the template argument list is explicitly written immediately following the class template name. For primary templates, this list is implicitly described by the template parameter list. Specifically, the order of the template arguments is the sequence in which they appear in the template parameter list. [Example: the template argument list for the primary template in the example above is <T1, T2, I>. — end example] [Note: The template argument list shall not be specified in the primary template declaration. For example,

§ 14.5.5

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:

```
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]

<sup>6</sup> 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:

- end example]

- <sup>7</sup> A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.
- Within the argument list of a class template partial specialization, the following restrictions apply:
  - A partially specialized non-type argument expression shall not involve a template parameter of the partial specialization except when the argument expression is a simple *identifier*. [Example:

```
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:

§ 14.5.5

```
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. 140
- 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]

- When a class template is used in a context that requires an instantiation of the class, it is necessary to determine whether the instantiation is to be generated using the primary template or one of the partial specializations. This is done by matching the template arguments of the class template specialization with the template argument lists of the partial specializations.
  - If exactly one matching specialization is found, the instantiation is generated from that specialization.
  - If more than one matching specialization is found, the partial order rules (14.5.5.2) are used to determine whether one of the specializations is more specialized than the others. If none of the specializations is more specialized than all of the other matching specializations, then the use of the class template is ambiguous and the program is ill-formed.
  - If no matches are found, the instantiation is generated from the primary template.
- A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (14.8.2). [Example:

- end example]
- A non-type template argument can also be deduced from the value of an actual template argument of a non-type parameter of the primary template. [Example: the declaration of a2 above. end example]
- In a type name that refers to a class template specialization, (e.g., A<int, int, 1>) the argument list shall match the template parameter list of the primary template. The template arguments of a specialization are deduced from the arguments of the primary template.

# 14.5.5.2 Partial ordering of class template specializations

[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

140) There is no way in which they could be used.

§ 14.5.5.2 325

— 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.

<sup>2</sup> [Example:

The partial specialization #2 is more specialized than the partial specialization #1 because the function template B is more specialized than the function template A according to the ordering rules for function templates. —  $end\ example$ ]

#### 14.5.5.3 Members of class template specializations

### [temp.class.spec.mfunc]

The template parameter list of a member of a class template partial specialization shall match the template parameter list of the class template partial specialization. The template argument list of a member of a class template partial specialization shall match the template argument list of the class template partial specialization. A class template specialization is a distinct template. The members of the class template partial specialization are unrelated to the members of the primary template. Class template partial specialization members that are used in a way that requires a definition shall be defined; the definitions of members of the primary template are never used as definitions for members of a class template partial specialization. An explicit specialization of a member of a class template partial specialization is declared in the same way as an explicit specialization of the primary template. [Example:

```
// primary template
template < class T, int I > struct A {
  void f();
};
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;
                                    // OK, uses definition of primary template's member
  a0.f();
                                    // OK, uses definition of
  a2.g();
                                    // partial specialization's member
                                    // OK, uses definition of
  a2.h();
```

§ 14.5.5.3

```
// explicit specialization's member
a2.f(); // ill-formed, no definition of f for A<T,2>
// the primary template is not used here
}
```

— end example]

If a member template of a class template is partially specialized, the member template partial specializations are member templates of the enclosing class template; if the enclosing class template is instantiated (14.7.1, 14.7.2), a declaration for every member template partial specialization is also instantiated as part of creating the members of the class template specialization. If the primary member template is explicitly specialized for a given (implicit) specialization of the enclosing class template, the partial specializations of the member template are ignored for this specialization of the enclosing class template. If a partial specialization of the member template is explicitly specialized for a given (implicit) specialization of the enclosing class template, the primary member template and its other partial specializations are still considered for this specialization of the enclosing class template. [Example:

# 14.5.6 Function templates

[temp.fct]

A function template defines an unbounded set of related functions. [Example: a family of sort functions might be declared like this:

```
template<class T> class Array { };
template<class T> void sort(Array<T>&);
```

— end example]

A function template can be overloaded with other function templates and with normal (non-template) functions. A normal function is not related to a function template (i.e., it is never considered to be a specialization), even if it has the same name and type as a potentially generated function template specialization.<sup>141</sup>

### 14.5.6.1 Function template overloading

[temp.over.link]

It is possible to overload function templates so that two different function template specializations have the same type. [Example:

- end example]

<sup>2</sup> Such specializations are distinct functions and do not violate the one definition rule (3.2).

§ 14.5.6.1 327

<sup>141)</sup> That is, declarations of non-template functions do not merely guide overload resolution of function template specializations with the same name. If such a non-template function is odr-used (3.2) in a program, it must be defined; it will not be implicitly instantiated using the function template definition.

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.

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:

-end note

- 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:

- end example] Two expressions involving template parameters that are not equivalent are functionally equivalent if, for any given set of template arguments, the evaluation of the expression results in the same value.
- Two function templates are equivalent if they are declared in the same scope, have the same name, have identical template parameter lists, and have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters. Two function templates are functionally equivalent if they are equivalent except that one or more expressions that involve template parameters in the return types and parameter lists are functionally equivalent using the rules described above to compare expressions involving template parameters. If a program contains declarations of function templates that are functionally equivalent but not equivalent, the program is ill-formed; no diagnostic is required.
- <sup>7</sup> [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>);
```

§ 14.5.6.1 328

```
// Ill-formed, no diagnostic required
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+1+2+3+4>);
— end note]
```

### 14.5.6.2 Partial ordering of function templates

[temp.func.order]

- If a function template is overloaded, the use of a function template specialization might be ambiguous because template argument deduction (14.8.2) may associate the function template specialization with more than one function template declaration. *Partial ordering* of overloaded function template declarations is used in the following contexts to select the function template to which a function template specialization refers:
  - during overload resolution for a call to a function template specialization (13.3.3);
  - when the address of a function template specialization is taken;
  - when a placement operator delete that is a function template specialization is selected to match a placement operator new (3.7.4.2, 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 type. The deduction process determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process.
- To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs (14.5.3) thereof) synthesize a unique type, value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template. If only one of the function templates is a non-static member, that function template is considered to have a new first parameter inserted in its function parameter list. The new parameter is of type "reference to cv A," where cv are the cv-qualifiers of the function template (if any) and A is the class of which the function template is a member. [Note: This allows a non-static member to be ordered with respect to a nonmember function and for the results to be equivalent to the ordering of two equivalent nonmembers. end note] [Example:

— end example]

<sup>4</sup> Using the transformed function template's function type, perform type deduction against the other template as described in 14.8.2.4.

[Example:

§ 14.5.6.2 329

```
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(const T*) is more specialized than f(T) or f(T*)
    f(p);
    float x;
                      // Ambiguous: g(T) or g(T&)
    g(x);
    A<int> z;
    h(z);
                      // overload resolution selects h(A<T>&)
    const A<int> z2;
                      // h(const T&) is called because h(A<T>&) is not callable
    h(z2);
 — end example]
[Note: Since partial ordering in a call context considers only parameters for which there are explicit call argu-
ments, some parameters are ignored (namely, function parameter packs, parameters with default arguments,
and ellipsis parameters). [Example:
  template<class T> void f(T);
                                           // #1
                                           // #2
  template<class T> void f(T*, int=1);
  template<class T> void g(T);
                                           // #3
  template<class T> void g(T*, ...);
                                           // #4
  int main() {
    int* ip;
                      // calls #2
    f(ip);
                      // calls #4
    g(ip);
 — end example ] [Example:
  template<class T, class U> struct A { };
  template<class T, class U> void f(U, A<U, T>* p = 0); // \#1
  template<
               class U> void f(U, A<U, U>* p = 0); // #2
                           > void g(T, T = T());
  template<class T
                                                      // #3
  template<class T, class... U> void g(T, U ...);
                                                          // #4
  void h() {
                                                          // calls #2
    f<int>(42, (A<int, int>*)0);
                                                          // error: ambiguous
    f<int>(42);
    g(42);
                                                          // error: ambiguous
— end example ] [Example:
  template<class T, class... U> void f(T, U...);
                                                          // #1
                             > void f(T);
                                                          // #2
  template<class T
```

§ 14.5.6.2

### 14.5.7 Alias templates

[temp.alias]

- A template-declaration in which the declaration is an alias-declaration (Clause 7) declares the identifier to be a alias template. An alias template is a name for a family of types. The name of the alias template is a template-name.
- When a template-id refers to the specialization of an alias template, it is equivalent to the associated type obtained by substitution of its template-arguments for the template-parameters in the type-id of the alias template. [Note: An alias template name is never deduced. end note] [Example:

```
template<class T> struct Alloc { /* ... */ };
template<class T> using Vec = vector<T, Alloc<T>>;
                     //\ same\ as\ {\tt vector}<{\tt int}, Alloc<int>> v;
Vec<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]
```

The *type-id* in an alias template declaration shall not refer to the alias template being declared. The type produced by an alias template specialization shall not directly or indirectly make use of that specialization. [Example:

### 14.6 Name resolution

[temp.res]

- Three kinds of names can be used within a template definition:
  - The name of the template itself, and names declared within the template itself.

- Names dependent on a template-parameter (14.6.2).
- Names from scopes which are visible within the template definition.

A name used in a template declaration or definition and that is dependent on a template-parameter is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword typename. [Example:

```
// 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;
                                      // declare pointer to T's A
     TA* a5;
      typename T::A* a6;
                                      // declare pointer to T's A
                                      // T:: A is not a type name:
     T::A* a7;
                                      // multiply T::A by a7; ill-formed,
                                      // no visible declaration of a7
                                      // B is not a type name:
     B* a8:
                                      // multiply B by a8; ill-formed,
                                      // no visible declarations of B and a8
   }
 };
- end example]
```

When a qualified-id is intended to refer to a type that is not a member of the current instantiation (14.6.2.1) and its nested-name-specifier refers to a dependent type, it shall be prefixed by the keyword typename, forming a typename-specifier. If the qualified-id in a typename-specifier does not denote a type, the program is ill-formed.

typename-specifier:

 $\verb|typename| nested-name-specifier| identifier|$ 

 $\verb|typename| nested-name-specifier| \verb|template| opt| simple-template-id|$ 

If a specialization of a template is instantiated for a set of template-arguments such that the qualified-id prefixed by typename does not denote a type, the specialization is ill-formed. The usual qualified name lookup (3.4.3) is used to find the qualified-id even in the presence of typename. [Example:

```
struct A {
   struct X { };
   int X;
};
struct B {
   struct X { };
};
template < class T> void f(T t) {
   typename T::X x;
}
void foo() {
   A a;
```

```
B b;
f(b); // OK: T::X refers to B::X
f(a); // error: T::X refers to the data member A::X not the struct A::X
}
```

— end example]

- A qualified name used as the name in a mem-initializer-id, a base-specifier, or an elaborated-type-specifier is implicitly assumed to name a type, without the use of the typename keyword. In a nested-name-specifier that immediately contains a nested-name-specifier that depends on a template parameter, the identifier or simple-template-id is implicitly assumed to name a type, without the use of the typename keyword. [Note: The typename keyword is not permitted by the syntax of these constructs. end note]
- <sup>6</sup> If, for a given set of template arguments, a specialization of a template is instantiated that refers to a qualified-id that denotes a type, and the qualified-id refers to a member of an unknown specialization, the qualified-id shall either be prefixed by typename or shall be used in a context in which it implicitly names a type as described above. [Example:

— end example]

Within the definition of a class template or within the definition of a member of a class template following the declarator-id, the keyword typename is not required when referring to the name of a previously declared member of the class template that declares a type. [Note: such names can be found using unqualified name lookup (3.4.1), class member lookup (3.4.3.1) into the current instantiation (14.6.2.1), or class member access expression lookup (3.4.5) when the type of the object expression is the current instantiation (14.6.2.2). — end note] [Example:

— end example]

8 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 every valid specialization of a variadic template requires an empty template parameter pack, 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) {
                       // diagnosed if X::f is instantiated
                       // and the assignment to t is an error
                       // may be diagnosed even if X::f is
     p = i;
                       // not instantiated
                       // may be diagnosed even if X::f is
     p = j;
                       // not instantiated
   }
   void g(T t) {
                       // may be diagnosed even if X::g is
     +;
                       // not instantiated
   }
 };
 template<class... T> struct A {
   void operator++(int, T... t);
                                                     // error: too many parameters
 };
                                                     // error: union with base class
 template<class... T> union X : T... { };
 template<class... T> struct A : T..., T... { };// error: duplicate\ base\ class
— end example]
```

When looking for the declaration of a name used in a template definition, the usual lookup rules (3.4.1, 3.4.2) are used for non-dependent names. The lookup of names dependent on the template parameters is postponed until the actual template argument is known (14.6.2). [Example:

```
#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';
    }
};</pre>
```

in the example, i is the local variable i declared in printal1, cnt is the member cnt declared in Set, and cout is the standard output stream declared in iostream. However, not every declaration can be found this way; the resolution of some names must be postponed until the actual template-arguments are known. For example, even though the name operator<< is known within the definition of printal1() 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:

§ 14.6

```
void f(char);
 template<class T> void g(T t) {
                       // f(char)
   f(1):
   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() {
                       // will cause one call of f(char) followed
   g(e);
                       // by two calls of f(E)
   g('a');
                       // will cause three calls of f(char)
— end example]
```

<sup>11</sup> [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 as a template-name or a type-name. When it is used with a template-argument-list, as a template-argument for a template template-parameter, or as the final identifier in the elaborated-type-specifier of a friend class template declaration, it refers to the class template itself. Otherwise, it is equivalent to the template-name followed by the template-parameters of the class template enclosed in <>.
- Within the scope of a class template specialization or partial specialization, when the injected-class-name is used as a *type-name*, it is equivalent to the *template-name* followed by the *template-arguments* of the class template specialization or partial specialization enclosed in <>. [Example:

— end example]

The injected-class-name of a class template or class template.

The injected-class-name of a class template or class template specialization can be used either as a template-name or a type-name wherever it is in scope. [Example:

```
template <class T> struct Base {
   Base* p;
};

template <class T> struct Derived: public Base<T> {
   typename Derived::Base* p; // meaning Derived::Base<T> };
```

<sup>4</sup> A lookup that finds an injected-class-name (10.2) can result in an ambiguity in certain cases (for example, if it is found in more than one base class). If all of the injected-class-names that are found refer to specializations of the same class template, and if the name is used as a *template-name*, the reference refers to the class template itself and not a specialization thereof, and is not ambiguous. [Example:

When the normal name of the template (i.e., the name from the enclosing scope, not the injected-class-name) is used, it always refers to the class template itself and not a specialization of the template. [Example:

<sup>6</sup> 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:

In the definition of a member of a class template that appears outside of the class template definition, the name of a member of the class template hides the name of a template-parameter of any enclosing class templates (but not a template-parameter of the member if the member is a class or function template). [Example:

<sup>8</sup> 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:

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:

#### 14.6.2 Dependent names

[temp.dep]

Inside a template, some constructs have semantics which may differ from one instantiation to another. Such a construct depends on the template parameters. In particular, types and expressions may depend on the type and/or value of template parameters (as determined by the template arguments) and this determines the context for name lookup for certain names. Expressions may be type-dependent (on the type of a template parameter) or value-dependent (on the value of a non-type template parameter). In an expression of the form:

```
postfix-expression ( expression-list_{opt})
```

where the postfix-expression is an id-expression, the id-expression denotes a dependent name if

- any of the expressions in the expression-list is a pack expansion (14.5.3),
- any of the expressions in the expression-list is a type-dependent expression (14.6.2.2), or
- if the *unqualified-id* of the *id-expression* is a *template-id* in which any of the template arguments depends on a template parameter.

If an operand of an operator is a type-dependent expression, the operator also denotes a dependent name. Such names are unbound and are looked up at the point of the template instantiation (14.6.4.1) in both the context of the template definition and the context of the point of instantiation.

<sup>2</sup> [Example:

```
template<class T> struct X : B<T> {
  typename T::A* pa;
  void f(B<T>* pb) {
    static int i = B<T>::i;
    pb->j++;
  }
};
```

the base class name B<T>, the type name T::A, the names B<T>::i and pb->j explicitly depend on the template-parameter. —  $end\ example$ ]

<sup>3</sup> 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:

The type name A in the definition of X<T> binds to the type def name defined in the global namespace scope, not to the type def name defined in the base class B<T>. — end example] [Example:

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]

- A name refers to the *current instantiation* if it is
  - in the definition of a class template, a nested class of a class template, a member of a class template, or a member of a nested class of a class template, the injected-class-name (Clause 9) of the class template or nested class,
  - in the definition of a primary class template or a member of a primary class template, the name of the class template followed by the template argument list of the primary template (as described below) enclosed in <> (or an equivalent template alias specialization),

§ 14.6.2.1 338

— 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 or a member of a partial specialization, the name of the class template followed by the template argument list of the partial specialization enclosed in <> (or an equivalent template alias specialization). If the nth template parameter is a parameter pack, the nth template argument is a pack expansion (14.5.3) whose pattern is the name of the parameter pack.
- <sup>2</sup> The template argument list of a primary template is a template argument list in which the *n*th template argument has the value of the *n*th template parameter of the class template. If the *n*th template parameter is a template parameter pack (14.5.3), the *n*th template argument is a pack expansion (14.5.3) whose pattern is the name of the template parameter pack.
- <sup>3</sup> A template argument that is equivalent to a template parameter (i.e., has the same constant value or the same type as the template parameter) can be used in place of that template parameter in a reference to the current instantiation. In the case of a non-type template argument, the argument must have been given the value of the template parameter and not an expression in which the template parameter appears as a subexpression. [Example:

```
template <class T> class A {
                                     // A is the current instantiation
   A* p1;
   A<T>* p2;
                                     // A<T> is the current instantiation
                                     // A<T*> is not the current instantiation
   A<T*> p3;
                                     // :: A<T> is the current instantiation
   ::A<T>* p4;
   class B {
     B* p1;
                                     // B is the current instantiation
     A < T > : : B * p2;
                                     // A<T>::B is the current instantiation
                                     // A<T*>::B is not the
      typename A<T*>::B* p3;
                                     // current instantiation
   };
 };
 template <class T> class A<T*> {
                                     // A<T*> is the current instantiation
   A < T *> * p1;
   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;
                                     // refers to the current instantiation
   B<my_T1, T2, my_I>* b3;
                                     // not the current instantiation
   B<my_T1, T2, my_I2>* b4;
   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 at least one member of the current instantiation or a non-dependent base class thereof. [Note: This can only occur when looking up a name in a scope enclosed by the definition of a class template. —end note]

§ 14.6.2.1

— A qualified-id in which the nested-name-specifier refers to the current instantiation and that, when looked up, refers to at least one member of the current instantiation or a non-dependent base class thereof. [Note: if no such member is found, and the current instantiation has any dependent base classes, then the qualified-id is a member of an unknown specialization; see below. —end note]

— An *id-expression* denoting the member in a class member access expression (5.2.5) for which the type of the object expression is the current instantiation, and the *id-expression*, when looked up (3.4.5), refers to at least one member of the current instantiation or a non-dependent base class thereof. [Note: if no such member is found, and the current instantiation has any dependent base classes, then the *id-expression* is a member of an unknown specialization; see below. — end note]

[Example:

- <sup>5</sup> A name is a member of an unknown specialization if it is
  - A qualified-id in which the nested-name-specifier names a dependent type that is not the current instantiation.
  - A qualified-id in which the nested-name-specifier refers to the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the qualified-id does not find any member of the current instantiation or a non-dependent base class thereof.
  - An id-expression denoting the member in a class member access expression (5.2.5) in which either
    - the type of the object expression is the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the *id-expression* does not find a member of the current instantiation or a non-dependent base class thereof; or
    - the type of the object expression is dependent and is not the current instantiation.
- <sup>6</sup> If a qualified-id in which the nested-name-specifier refers to the current instantiation is not a member of the current instantiation or a member of an unknown specialization, the program is ill-formed even if the template containing the qualified-id is not instantiated; no diagnostic required. Similarly, if the id-expression in a class member access expression for which the type of the object expression is the current instantiation does not refer to a member of the current instantiation or a member of an unknown specialization, the program is ill-formed even if the template containing the member access expression is not instantiated; no diagnostic required. [Example:

§ 14.6.2.1 340

```
/\!/\;a\;member\;of\;an\;unknown\;specialization\\ \} };
```

- end example]
- If, for a given set of template arguments, a specialization of a template is instantiated that refers to a member of the current instantiation with a qualified-id or class member access expression, the name in the qualified-id or class member access expression is looked up in the template instantiation context. If the result of this lookup differs from the result of name lookup in the template definition context, name lookup is ambiguous. [Note: the result of name lookup differs only when the member of the current instantiation was found in a non-dependent base class of the current instantiation and a member with the same name is also introduced by the substitution for a dependent base class of the current instantiation. end note]
- <sup>8</sup> A type is dependent if it is
  - a template parameter,
  - a member of an unknown specialization,
  - a nested class or enumeration 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,
  - 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

[temp.dep.expr]

- <sup>1</sup> Except as described below, an expression is type-dependent if any subexpression is type-dependent.
- <sup>2</sup> 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:

§ 14.6.2.2 341

```
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 \\ \end{cases}
```

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
  noexcept ( expression )

[ Note: For the standard library macro offsetof, see 18.2. — end note ]
```

A class member access expression (5.2.5) is type-dependent if the expression refers to a member of the current instantiation and the type of the referenced member is dependent, or the class member access expression refers to a member of an unknown specialization. [Note: In an expression of the form 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]

- Except as described below, a constant expression is value-dependent if any subexpression is value-dependent.
- <sup>2</sup> 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)
typeid (expression)
typeid (type-id)
alignof (type-id)
noexcept (expression)

[Note: For the standard library macro offsetof, see 18.2.—end note]
```

<sup>3</sup> 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:

§ 14.6.2.3

```
simple-type-specifier (expression-list_{opt}) \\ \texttt{static\_cast} < type-id > (expression) \\ \texttt{const\_cast} < type-id > (expression) \\ \texttt{reinterpret\_cast} < type-id > (expression) \\ (type-id) cast-expression \\ \end{cases}
```

Expressions of the following form are value-dependent:

```
sizeof ... ( identifier )
```

<sup>5</sup> An *id-expression* is value-dependent if it names a member of an unknown specialization.

#### 14.6.2.4 Dependent template arguments

[temp.dep.temp]

- A type template-argument is dependent if the type it specifies is dependent.
- <sup>2</sup> A non-type template-argument is dependent if its type is dependent or the constant expression it specifies is value-dependent.
- <sup>3</sup> Furthermore, a non-type template-argument is dependent if the corresponding non-type template-parameter is of reference or pointer type and the template-argument designates or points to a member of the current instantiation or a member of a dependent type.
- <sup>4</sup> A template template-argument is dependent if it names a template-parameter or is a qualified-id that refers to a member of an unknown specialization.

### 14.6.3 Non-dependent names

[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:

```
void g(double);
 void h();
 template<class T> class Z {
 public:
    void f() {
                        // calls g(double)
      g(1);
                        // ill-formed: cannot increment function;
      h++;
                        // 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]

- <sup>1</sup> 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 for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

§ 14.6.4.1 343

<sup>2</sup> If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

- For a class template specialization, a class member template specialization, or a specialization for a class member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization, if the context from which the specialization is referenced depends on a template parameter, and if the specialization is not instantiated previous to the instantiation of the enclosing template, the point of instantiation is immediately before the point of instantiation of the enclosing template. Otherwise, the point of instantiation for such a specialization immediately precedes the namespace scope declaration or definition that refers to the specialization.
- <sup>4</sup> If a virtual function is implicitly instantiated, its point of instantiation is immediately following the point of instantiation of its enclosing class template specialization.
- <sup>5</sup> An explicit instantiation definition is an instantiation point for the specialization or specializations specified by the explicit instantiation.
- 6 The instantiation context of an expression that depends on the template arguments is the set of declarations with external linkage declared prior to the point of instantiation of the template specialization in the same translation unit.
- A specialization for a function template, a member function template, or of a member function or static data member of a class template may have multiple points of instantiations within a translation unit, and in addition to the points of instantiation described above, for any such specialization that has a point of instantiation within the translation unit, the end of the translation unit is also considered a point of instantiation. A specialization for a class template has at most one point of instantiation within a translation unit. A specialization for any template may have points of instantiation in multiple translation units. If two different points of instantiation give a template specialization different meanings according to the one definition rule (3.2), the program is ill-formed, no diagnostic required.

#### 14.6.4.2 Candidate functions

# [temp.dep.candidate]

- For a function call that depends on a template parameter, the candidate functions are found using the usual lookup rules (3.4.1, 3.4.2, 3.4.3) except that:
  - For the part of the lookup using unqualified name lookup (3.4.1) or qualified name lookup (3.4.3), 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 function name is an *unqualified-id* and 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

|temp.inject|

- <sup>1</sup> Friend classes or functions can be declared within a class template. When a template is instantiated, the names of its friends are treated as if the specialization had been explicitly declared at its point of instantiation.
- As with non-template classes, the names of namespace-scope friend functions of a class template specialization are not visible during an ordinary lookup unless explicitly declared at namespace scope (11.3). Such names may be found under the rules for associated classes (3.4.2).<sup>142</sup> [Example:

```
template<typename T> struct number {
  number(int);
  friend number gcd(number x, number y) { return 0; };
```

<sup>142)</sup> Friend declarations do not introduce new names into any scope, either when the template is declared or when it is instantiated.

# 14.7 Template instantiation and specialization

[temp.spec]

- <sup>1</sup> The act of instantiating a function, a class, a member of a class template or a member template is referred to as template instantiation.
- A function instantiated from a function template is called an instantiated function. A class instantiated from a class template is called an instantiated class. A member function, a member class, a member enumeration, or a static data member of a class template instantiated from the member definition of the class template is called, respectively, an instantiated member function, member class, member enumeration, or static data member. A member function instantiated from a member function template is called an instantiated member function. A member class instantiated from a member class template is called an instantiated member class.
- An explicit specialization may be declared for a function template, a class template, a member of a class template or a member template. An explicit specialization declaration is introduced by template<>. In an explicit specialization declaration for a class template, a member of a class template or a class member template, the name of the class that is explicitly specialized shall be a simple-template-id. In the explicit specialization declaration for a function template or a member function template, the name of the function or member function explicitly specialized may be a template-id. [Example:

```
template<class T = int> struct A {
   static int x;
 };
 template < class U > void g(U) { }
                                            // specialize for T == double
 template<> struct A<double> { };
                                            // specialize for T == int
 template<> struct A<> { };
 template<> void g(char) { }
                                            // specialize for U == char
                                            // U is deduced from the parameter type
 template<> void g<int>(int) { }
                                            // specialize for U == int
 template<> int A<char>::x = 0;
                                            // specialize for T == char
 template<class T = int> struct B {
   static int x;
 template<> int B<>::x = 1;
                                            // specialize for T == int
— end example]
```

- <sup>4</sup> An instantiated template specialization can be either implicitly instantiated (14.7.1) for a given argument list or be explicitly instantiated (14.7.2). A specialization is a class, function, or class member that is either instantiated or explicitly specialized (14.7.3).
- For a given template and a given set of template-arguments,
  - an explicit instantiation definition shall appear at most once in a program,
  - an explicit specialization shall be defined at most once in a program (according to 3.2), and
  - both an explicit instantiation and a declaration of an explicit specialization shall not appear in a program unless the explicit instantiation follows a declaration of the explicit specialization.

§ 14.7 345

An implementation is not required to diagnose a violation of this rule.

Each class template specialization instantiated from a template has its own copy of any static members. [Example:

```
template<class T> class X {
   static T s;
};
template<class T> T X<T>::s = 0;
X<int> aa;
X<char*> bb;
```

X<int> has a static member s of type int and X<char\*> has a static member s of type char\*. — end example

# 14.7.1 Implicit instantiation

[temp.inst]

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, scoped member enumerations, static data members and member templates; and it causes the implicit instantiation of the definitions of unscoped member enumerations and member anonymous unions. However, for the purpose of determining whether an instantiated redeclaration of a member is valid according to 9.2, a declaration that corresponds to a definition in the template is considered to be a definition. [Example:

Outer<int, int>::Inner<int, Y> is redeclared at #1b. (It is not defined but noted as being associated with a definition in Outer<T, U>.) #2 is also a redeclaration of #1a. It is noted as associated with a definition, so it is an invalid redeclaration of the same partial specialization. — end example]

- <sup>2</sup> Unless a member of a class template or a member template has been explicitly instantiated or explicitly specialized, the specialization of the member is implicitly instantiated when the specialization is referenced in a context that requires the member definition to exist; in particular, the initialization (and any associated side-effects) of a static data member does not occur unless the static data member is itself used in a way that requires the definition of the static data member to exist.
- Unless a function template specialization has been explicitly instantiated or explicitly specialized, the function template specialization is implicitly instantiated when the specialization is referenced in a context that requires a function definition to exist. Unless a call is to a function template explicit specialization or to a member function of an explicitly specialized class template, a default argument for a function template or a member function of a class template is implicitly instantiated when the function is called in a context that requires the value of the default argument.
- 4 [Example:

```
template<class T> struct Z {
  void f();
  void g();
```

§ 14.7.1

Nothing in this example requires class Z<double>, Z<int>::g(), or Z<char>::f() to be implicitly instantiated. — end example

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:

<sup>6</sup> 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:

<sup>7</sup> If an implicit instantiation of a class template specialization is required and the template is declared but not defined, the program is ill-formed. [Example:

```
template<class T> class X;
```

§ 14.7.1 347

```
X<char> ch; // error: definition of X required
```

— end example]

8 The implicit instantiation of a class template does not cause any static data members of that class to be implicitly instantiated.

- <sup>9</sup> 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:

```
namespace N {
  template<class T> class List {
  public:
    T* get();
  };
}

template<class K, class V> class Map {
  public:
    N::List<V> lt;
    V get(K);
};

void g(Map<const char*,int>& m) {
  int i = m.get("Nicholas");
}
```

a call of lt.get() from Map<const char\*,int>::get() would place List<int>::get() in the name-space N rather than in the global namespace. — end example

If a function template f is called in a way that requires a default argument to be used, the dependent names are looked up, the semantics constraints are checked, and the instantiation of any template used in the default argument is done as if the default argument had been an initializer used in a function template specialization with the same scope, the same template parameters and the same access as that of the function template f used at that point. 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:

```
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) {
```

§ 14.7.1 348

```
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]
```

- <sup>14</sup> [Note: 14.6.4.1 defines the point of instantiation of a template specialization. end note]
- There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations, which could involve more than one template. The result of an infinite recursion in instantiation is undefined. [Example:

#### 14.7.2 Explicit instantiation

[temp.explicit]

- A class, a function or member template specialization can be explicitly instantiated from its template. A member function, member class or static data member of a class template can be explicitly instantiated from the member definition associated with its class template. An explicit instantiation of a function template or member function of a class template shall not use the inline or constexpr specifiers.
- The syntax for explicit instantiation is:

 $explicit\mbox{-}instantiation:$ 

 $extern_{opt}$  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:

<sup>4</sup> A declaration of a function template, a member function or static data member of a class template, or a member function template of a class or class template shall precede an explicit instantiation of that entity. A definition of a class template, a member class of a class template, or a member class template of a class or class template shall precede an explicit instantiation of that entity unless the explicit instantiation is preceded by an explicit specialization of the entity with the same template arguments. If the *declaration* of the explicit instantiation names an implicitly-declared special member function (Clause 12), the program is ill-formed.

- For a given set of template arguments, if an explicit instantiation of a template appears after a declaration of an explicit specialization for that template, the explicit instantiation has no effect. Otherwise, for an explicit instantiation definition the definition of a function template, a member function template, or a member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.
- 6 An explicit instantiation of a class or function template specialization is placed in the namespace in which the template is defined. An explicit instantiation for a member of a class template is placed in the namespace where the enclosing class template is defined. An explicit instantiation for a member template is placed in the namespace where the enclosing class or class template is defined. [Example:

```
namespace N {
   template<class T> class Y { void mf() { } };
 }
 template class Y<int>;
                                             // error: class template Y not visible
                                             // in the global namespace
 using N::Y;
                                             // error: explicit instantiation outside of the
 template class Y<int>;
                                              // namespace of the template
                                             // OK: explicit instantiation in namespace N
 template class N::Y<char*>;
                                             // OK: explicit instantiation
 template void N::Y<double>::mf();
                                              // in namespace N
— end example]
```

A trailing template-argument can be left unspecified in an explicit instantiation of a function template specialization or of a member function template specialization provided it can be deduced from the type of a function parameter (14.8.2). [Example:

```
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

// instantiate sort(Array<int>&) - template-argument deduced
template void sort<>(Array<int>&);
```

- end example]
- An explicit instantiation that names a class template specialization is 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]
- 9 An explicit instantiation definition that names a class template specialization explicitly instantiates the class template specialization and is an explicit instantiation definition of only those members that have been defined at the point of instantiation.
- 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 odr-used (3.2) so that the body can be considered for inlining, but that no out-of-line copy of the inline function would be generated in the translation unit. — end note

- If an entity is the subject of both an explicit instantiation declaration and an explicit instantiation definition in the same translation unit, the definition shall follow the declaration. An entity that is the subject of an explicit instantiation declaration and that is also used in a way that would otherwise cause an implicit instantiation (14.7.1) in the translation unit shall be the subject of an explicit instantiation definition 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:

```
char* p = 0;
template<class T> T g(T x = &p) { return x; }
template int g<int>(int);  // OK even though &p isn't an int.

— end example]
```

#### 14.7.3 Explicit specialization

[temp.expl.spec]

- An explicit specialization of any of the following:
  - function template
  - class template
  - member function of a class template
  - static data member of a class template
  - member class of a class template
  - member enumeration of a class template
  - member class template of a class or class template
  - member function template of a class or class template

can be declared by a declaration introduced by template<>; that is: explicit-specialization: template <> declaration

```
[Example:

template<class T> class stream;

template<> class stream<char> { /* ... */ };

template<class T> class Array { /* ... */ };

template<class T> void sort(Array<T>& v) { /* ... */ }

template<> void sort<char*>(Array<char*>&);
```

Given these declarations, stream<char> will be used as the definition of streams of chars; other streams will be handled by class template specializations instantiated from the class template. Similarly, sort<char\*> will be used as the sort function for arguments of type Array<char\*>; other Array types will be sorted by functions generated from the template. — end example]

- An explicit specialization shall be declared in a namespace enclosing the specialized template. An explicit specialization whose *declarator-id* is not qualified shall be declared in the nearest enclosing namespace of the template, or, if the namespace is inline (7.3.1), any namespace from its enclosing namespace set. Such a declaration may also be a definition. If the declaration is not a definition, the specialization may be defined later (7.3.1.2).
- A declaration of a function template 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:

- <sup>4</sup> A member function, a member function template, a member class, a member enumeration, a member class template, 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.
- A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the class template; instead, the member of the class template specialization shall itself be explicitly defined if its definition is required. In this case, the definition of the class template explicit specialization shall be in scope at the point at which the member is defined. The definition of an explicitly specialized class is unrelated to the definition of a generated specialization. That is, its members need not have the same names, types, etc. as the members of a generated specialization. Members of an explicitly specialized class template are defined in the same manner as members of normal classes, and not using the template<> syntax. The same is true when defining a member of an explicitly specialized member class. However, template<> is used in defining a member of an explicitly specialized member class template that is specialized as a class template. [Example:

```
template<> struct A<char>::B {
   void f();
 };
 // template<> also not used when defining a member of
 // an explicitly specialized member class
 void A<char>::B::f() { /* ... */ }
 template<> template<class U> struct A<char>::C {
   void f();
 // template<> is used when defining a member of an explicitly
 // specialized member class template specialized as a class template
 template<class U> void A<char>::C<U>::f() { /* ... */ }
 template<> struct A<short>::B {
   void f();
 };
 template<> void A<short>::B::f() { /* ... */ } // error: template<> not permitted
 template<> template<class U> struct A<short>::C {
   void f();
};
 template<class U> void A<short>::C<U>::f() { /* ... */ } // error: template<> required
- end example]
```

If a template, a member template or a member of a class template is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs; no diagnostic is required. If the program does not provide a definition for an explicit specialization and either the specialization is used in a way that would cause an implicit instantiation to take place or the member is a virtual member function, the program is ill-formed, no diagnostic required. An implicit instantiation is never generated for an explicit specialization that is declared but not defined. [Example:

```
class String { };
template<class T> class Array { /* \dots */ };
template<class T> void sort(Array<T>& v) { /* ... */ }
void f(Array<String>& v) {
                     // use primary template
  sort(v);
                     // sort(Array<T>&), T is String
}
template<> void sort<String>(Array<String>& v); // error: specialization
                                                  // after use of primary template
template<> void sort<>(Array<char*>& v);
                                                  // OK: sort<char*> not yet used
template<class T> struct A {
  enum E : T;
  enum class S : T;
};
template<> enum A<int>::E : int { eint };
                                                    // OK
                                                    // OK
template<> enum class A<int>::S : int { sint };
template<class T> enum A<T>::E : T { eT };
template<class T> enum class A<T>::S : T { sT };
                                                    // ill-formed, A<char>::E was instantiated
template<> enum A<char>::E : int { echar };
```

```
// when A<char> was instantiated template<> enum class A<char>::S : int { schar }; // OK
— end example]
```

The placement of explicit specialization declarations for function templates, class templates, member functions of class templates, static data members of class templates, member classes of class templates, member function 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> { /* ... */ };

  template<> class Y<double>;

  template<> class Y<double>;

  template<> class N::Y<double> { /* ... */ };

  // OK: specialization
  // in same namespace
  // forward declare intent to
  // specialize for double
}

template<> class N::Y<double> { /* ... */ };

// OK: specialization
  // in same namespace

— end example]
```

<sup>9</sup> 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;
template<> class X<int>;

X<int>* p;
X<int> x;
// OK: pointer to declared class X<int>
// error: object of incomplete class X<int>
— end example]
```

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]
```

- cha campic
- 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).
- 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:

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:

A member or a member template of a class template may be explicitly specialized for a given implicit instantiation of the class template, even if the member or member template is defined in the class template definition. An explicit specialization of a member or member template is specified using the syntax for explicit specialization. [Example:

```
template < class T > struct A {
   void f(T);
   template<class X1> void g1(T, X1);
   template < class X2> void g2(T, X2);
   void h(T) { }
 };
 // specialization
 template<> void A<int>::f(int);
 // out of class member template definition
 template<class T> template<class X1> void A<T>::g1(T, X1) { }
 // member template specialization
 template<> template<class X1> void A<int>::g1(int, X1);
 //member template specialization
 template<> template<>
   void A<int>::g1(int, char);
                                            // X1 deduced as char
 template<> template<>
   void A<int>::g2<char>(int, char);
                                            // X2 specified as char
 // member specialization even if defined in class definition
 template<> void A<int>::h(int) { }
— end example]
```

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

[temp.fct.spec]

- <sup>1</sup> 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;
};
```

§ 14.8 356

Here f<int>(int\*) has a static variable s of type int and f<char\*>(char\*\*) has a static variable s of type char\*. — end example]

#### 14.8.1 Explicit template argument specification

[temp.arg.explicit]

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:

- end example]
- <sup>2</sup> 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.
- 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:

§ 14.8.1 357

<sup>4</sup> [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:

<sup>5</sup> 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:

— end example]

— end example]

Implicit conversions (Clause 4) will be performed on a function argument to convert it to the type of the corresponding function parameter if the parameter type contains no *template-parameters* that participate in template argument deduction. [Note: Template parameters do not participate in template argument deduction if they are explicitly specified. For example,

```
template<class T> void f(T);

class Complex {
    Complex(double);
};

void g() {
    f<Complex>(1);
}

— end note]
// OK, means f<Complex>(Complex(1))
```

- 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 ]
- <sup>8</sup> [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

§ 14.8.1 358

the correct syntactic form unless there is a function template with that name visible at the point of the call. If no such name is visible, the call is not syntactically well-formed and argument-dependent lookup does not apply. If some such name is visible, argument dependent lookup applies and additional function templates may be found in other namespaces. [Example:

```
namespace A {
   struct B { };
   template<int X> void f(B);
 namespace C {
   template<class T> void f(T t);
 void g(A::B b) {
   f < 3 > (b);
                                    // ill-formed: not a function call
   A::f<3>(b);
                                    // well-formed
   C::f<3>(b);
                                    // ill-formed; argument dependent lookup
                                    // applies only to unqualified names
   using C::f;
                                    // well-formed because C::f is visible; then
   f<3>(b);
                                    // A::f is found by argument dependent lookup
-end example - end note
```

Template argument deduction can extend the sequence of template arguments corresponding to a template parameter pack, even when the sequence contains explicitly specified template arguments. [Example:

```
template<class ... Types> void f(Types ... values);

void g() {
   f<int*, float*>(0, 0, 0);  // Types is deduced to the sequence int*, float*, int }

— end example]
```

## 14.8.2 Template argument deduction

— end example]

[temp.deduct]

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:

- 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:

```
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);

    // #4: function type is h(int, const int*) h<const int>(1,0);
}

— end example
```

- <sup>4</sup> [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:

-end example

When all template arguments have been deduced or obtained from default template arguments, all uses of template parameters in the template parameter list of the template and the function type are replaced

with the corresponding deduced or default argument values. If the substitution results in an invalid type, as described above, type deduction fails.

- 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.
- <sup>7</sup> 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. [Note: Access checking is done as part of the substitution process. end note] Only invalid types and expressions in the immediate context of the function type and its template parameter types can result in a deduction failure. [Note: The evaluation of the substituted types and expressions can result in side effects such as the instantiation of class template specializations and/or function template specializations, the generation of implicitly-defined functions, etc. Such side effects are not in the "immediate context" and can result in the program being ill-formed. end note]

```
[Example:
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:

— Attempting to use a type that is not a class or enumeration type in a qualified name. [Example:

```
template <class T> int f(typename T::B*);
int i = f<int>(0);

— end example]
```

— Attempting to use a type in a nested-name-specifier of a qualified-id when that type does not contain the specified member, or

- the specified member is not a type where a type is required, or
- the specified member is not a template where a template is required, or
- the specified member is not a non-type where a non-type is required.

#### [Example:

```
template <int I> struct X { };
 template <template <class T> class> struct Z { };
 template <class T> void f(typename T::Y*){}
 template <class T> void g(X<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]
```

— Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration. [Example:

```
— 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.

```
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

[temp.deduct.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  $\mathtt{std}::\mathtt{initializer\_list}< P'>$  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:

— 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:

- <sup>2</sup> 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.
- <sup>3</sup> 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:

- end example]
- <sup>4</sup> 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.

[Note: as specified in 14.8.1, implicit conversions will be performed on a function argument to convert it to the type of the corresponding function parameter if the parameter contains no template-parameters that participate in template argument deduction. Such conversions are also allowed, in addition to the ones described in the preceding list. —  $end\ note$ ]

- 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]
- <sup>6</sup> 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.

7 [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 \langle class T \rangle int f(T, T (*p)(T));
           char g(char);
           template <class T> T g(T);
           int i = f(1, g); // calls f(int, int (*)(int))
         — end example]
```

# 14.8.2.2 Deducing template arguments taking the address of a function template [temp.deduct.funcaddr]

Template arguments can be deduced from the type specified when taking the address of an overloaded function (13.4). The function template's function type and the specified type are used as the types of P and A, and the deduction is done as described in 14.8.2.5.

#### 14.8.2.3 Deducing conversion function template arguments

[temp.deduct.conv]

- Template argument deduction is done by comparing the return type of the conversion function template (call it P; 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.
- <sup>2</sup> 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 cy-qualified type, the top level cy-qualifiers of P's type are ignored for type deduction.
- <sup>4</sup> 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.
- <sup>6</sup> 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
```

```
cv_{1,0} "pointer to ..." cv_{1,n-1} "pointer to" cv_{1,n}T1 and P is a type cv_{2,0} "pointer to ..." cv_{2,n-1} "pointer to" cv_{2,n}T2
```

The cv-unqualified T1 and T2 are used as the types of A and P respectively for type deduction. [Example:

```
struct A {
   template <class T> operator T***();
};
A a;
const int * const * const * p1 = a;  // T is deduced as int, not const int

— end example]
```

#### 14.8.2.4 Deducing template arguments during partial ordering [temp.deduct.partial]

- 1 Template argument deduction is done by comparing certain types associated with the two function templates being compared.
- Two sets of types are used to determine the partial ordering. For each of the templates involved there is the original function type and the transformed function type. [Note: The creation of the transformed type is described in 14.5.6.2. end note] The deduction process uses the transformed type as the argument template and the original type of the other template as the parameter template. This process is done twice for each type involved in the partial ordering comparison: once using the transformed template-1 as the argument template and template-2 as the parameter template and again using the transformed template-2 as the argument template and template-1 as the parameter template.
- 3 The types used to determine the ordering depend on the context in which the partial ordering is done:
  - In the context of a function call, the types used are those function parameter types for which the function call has arguments. 143
  - 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.
- <sup>4</sup> Each type nominated above from the parameter template and the corresponding type from the argument template are used as the types of P and A.
- <sup>5</sup> Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:
  - If P is a reference type, P is replaced by the type referred to.

<sup>143)</sup> Default arguments are not considered to be arguments in this context; they only become arguments after a function has been selected.

- If A is a reference type, A is replaced by the type referred to.
- <sup>6</sup> 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.
- <sup>7</sup> Remove any top-level cy-qualifiers:
  - If P is a cv-qualified type, P is replaced by the cv-unqualified version of P.
  - If A is a cv-qualified type, A is replaced by the cv-unqualified version of A.
- <sup>8</sup> If A was transformed from a function parameter pack and P is not a parameter pack, type deduction fails. Otherwise, using the resulting types P and A, the deduction is then done as described in 14.8.2.5. If P is a function parameter pack, the type A of each remaining parameter type of the argument template is compared with the type P of the declarator-id of the function parameter pack. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. If deduction succeeds for a given type, the type from the argument template is considered to be at least as specialized as the type from the parameter template. [Example:

- end example]
- <sup>9</sup> If, for a given type, deduction succeeds in both directions (i.e., the types are identical after the transformations above) and both P and A were reference types (before being replaced with the type referred to above):
  - if the type from the argument template was an lvalue reference and the type from the parameter template was not, the argument type is considered to be more specialized than the other; otherwise,
  - if the type from the argument template is more cv-qualified than the type from the parameter template (as described above), the argument type is considered to be more specialized than the other; otherwise,
  - neither type is more specialized than the other.
- <sup>10</sup> If for each type being considered a given template is at least as specialized for all types and more specialized for some set of types and the other template is not more specialized for any types or is not at least as specialized for any types, then the given template is more specialized than the other template. Otherwise, neither template is more specialized than the other.
- In most cases, all template parameters must have values in order for deduction to succeed, but for partial ordering purposes a template parameter may remain without a value provided it is not used in the types being used for partial ordering. [Note: A template parameter used in a non-deduced context is considered used. end note] [Example:

```
— 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:

#### 14.8.2.5 Deducing template arguments from a type

[temp.deduct.type]

- Template arguments can be deduced in several different contexts, but in each case a type that is specified in terms of template parameters (call it P) is compared with an actual type (call it A), and an attempt is made to find template argument values (a type for a type parameter, a value for a non-type parameter, or a template for a template parameter) that will make P, after substitution of the deduced values (call it the deduced A), compatible with A.
- In some cases, the deduction is done using a single set of types P and A, in other cases, there will be a set of corresponding types P and A. Type deduction is done independently for each P/A pair, and the deduced template argument values are then combined. If type deduction cannot be done for any P/A pair, or if for any pair the deduction leads to more than one possible set of deduced values, or if different pairs yield different deduced values, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails.
- 3 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.
- 5 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:

```
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.
- When a type name is specified in a way that includes a non-deduced context, all of the types that comprise that type name are also non-deduced. However, a compound type can include both deduced and non-deduced types. [Example: If a type is specified as A<T>::B<T2>, both T and T2 are non-deduced. Likewise, if a type is specified as A<I+J>::X<T>, I, J, and T are non-deduced. If a type is specified as 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]
- <sup>7</sup> [Example: Here is an example in which different parameter/argument pairs produce inconsistent template argument deductions:

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:

Here is an example where a qualification conversion applies between the argument type on the function call and the deduced template argument type:

Here is an example where the template argument is used to instantiate a derived class type of the corresponding function parameter type:

— end example]

<sup>8</sup> 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:

```
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 \quad (type ::*)(T)
type (T::*)(T)
T (type ::*)(T)
T (T::*)()
T (T::*)(T)
type [i]
template-name <i> (where template-name refers to a class template)
TT<T>
TT<i>
TT<>
```

where (T) represents a parameter-type-list where at least one parameter type contains a T, and () represents a parameter-type-list where no parameter type contains a T. Similarly, <T> represents template argument lists where at least one argument contains a T, <i> represents template argument lists where at least one argument contains an i and <> represents template argument lists where no argument contains a T or an i.

<sup>9</sup> If P has a form that contains <T> or <i>, then each argument P<sub>i</sub> of the respective template argument list P is compared with the corresponding argument A<sub>i</sub> of the corresponding template argument list of A. If

the template argument list of P contains a pack expansion that is not the last template argument, the entire template argument list is a non-deduced context. If  $P_i$  is a pack expansion, then the pattern of  $P_i$  is compared with each remaining argument in the template argument list of A. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by  $P_i$ . During partial ordering (14.8.2.4), if  $A_i$  was originally a pack expansion:

- if P does not contain a template argument corresponding to  $A_i$  then  $A_i$  is ignored;
- otherwise, if  $P_i$  is not a pack expansion, template argument deduction fails.

#### [Example:

— end example]

Similarly, if P has a form that contains (T), then each parameter type  $P_i$  of the respective parameter-type-list of P is compared with the corresponding parameter type  $A_i$  of the corresponding parameter-type-list of A. If P and A are function types that originated from deduction when taking the address of a function template (14.8.2.2) or when deducing template arguments from a function declaration (14.8.2.6) and  $P_i$  and  $A_i$  are parameters of the top-level parameter-type-list of P and A, respectively,  $P_i$  is adjusted if it is an realue reference to a cv-unqualified template parameter and  $A_i$  is an Ivalue reference, in which case the type of  $P_i$  is changed to be the template parameter type (i.e., T&& is changed to simply T). [Note: As a result, when  $P_i$  is T&& and  $A_i$  is X&, the adjusted  $P_i$  will be T, causing T to be deduced as X&. — end note] [Example:

```
template <class T> void f(T&&); template <> void f(int&) { } // #1 template <> void f(int&&) { } // #2 void g(int i) { // calls f<int&>(int&), i.e., #1 f(0); // calls f<int>(int&&), i.e., #2 }
```

— end example]

If the parameter-declaration corresponding to  $P_i$  is a function parameter pack, then the type of its declarator-id is compared with each remaining parameter type in the parameter-type-list of A. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. During partial ordering (14.8.2.4), if  $A_i$  was originally a function parameter pack:

- if P does not contain a function parameter type corresponding to  $A_i$  then  $A_i$  is ignored;
- otherwise, if  $P_i$  is not a function parameter pack, template argument deduction fails.

#### [Example:

```
template<class T, class... U> void f(T*, U...) { } // #1 template<class T> void f(T) { } // #2 template void f(int*); // selects #1
```

```
— end example]
```

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]
```

Template arguments cannot be deduced from function arguments involving constructs other than the ones specified above.

- <sup>13</sup> A template type argument cannot be deduced from the type of a non-type template-argument.
- 14 [Example:

<sup>15</sup> [ 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:

<sup>16</sup> 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:

— 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,

```
// but T is deduced here
        Tt,
        typename B<i>::Y y);
                                        // i is not deduced here
      A<int> a;
      B<77> b;
             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. 144 [Example:
      template<int i> class A { /* ... */ };
      template<short s> void f(A<s>);
      void k1() {
        A<1>a;
                           // error: deduction fails for conversion from int to short
        f(a);
        f<1>(a);
                           //OK
      template<const short cs> class B { };
      template<short s> void g(B<s>);
      void k2() {
        B<1> b:
        g(b);
                           // OK: cv-qualifiers are ignored on template parameter types
      }
    — end example]
   A template-argument can be deduced from a function, pointer to function, or pointer to member function
```

type.

[Example:

```
template<class T> void f(void(*)(T,int));
template<class T> void foo(T,int);
void g(int,int);
void g(char,int);
void h(int,int,int);
void h(char,int);
int m() {
                     // error: ambiguous
  f(&g);
  f(&h);
                     // OK: void h(char,int) is a unique match
                     // error: type deduction fails because foo is a template
 f(&foo);
```

— end example]

A template type-parameter cannot be deduced from the type of a function default argument. [Example:

```
template \langle class\ T \rangle void f(T = 5, T = 7);
```

<sup>144)</sup> 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.

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:

[Note: Template argument deduction involving parameter packs (14.5.3) can deduce zero or more arguments for each parameter pack. — end note] [Example:

```
template<class> struct X { };
template<class R, class ... ArgTypes> struct X<R(int, ArgTypes ...)> { };
template<class ... Types> struct Y { };
template<class T, class ... Types> struct Y<T, Types& ...> { };
template<class ... Types> int f(void (*)(Types ...));
void g(int, float);
X<int> x1;
                                  // uses primary template
X<int(int, float, double) > x2; // uses partial specialization; ArgTypes contains float, double
                                  // uses primary template
X<int(float, int)> x3;
Y<> y1;
                                  // use primary template; Types is empty
                                  // uses partial specialization; T is int&, Types contains float, double
Y<int&, float&, double&> y2;
Y<int, float, double> y3;
                                  // uses primary template; Types contains int, float, double
int fv = f(g);
                                  // OK; Types contains int, float
  — end example]
```

#### 14.8.2.6 Deducing template arguments from a function declaration [temp.deduct.decl]

- In a declaration whose declarator-id refers to a specialization of a function template, template argument deduction is performed to identify the specialization to which the declaration refers. Specifically, this is done for explicit instantiations (14.7.2), explicit specializations (14.7.3), and certain friend declarations (14.5.4). This is also done to determine whether a deallocation function template specialization matches a placement operator new (3.7.4.2, 5.3.4). In all these cases, P is the type of the function template being considered as a potential match and A is either the function type from the declaration or the type of the deallocation function that would match the placement operator new as described in 5.3.4. The deduction is done as described in 14.8.2.5.
- If, for the set of function templates so considered, there is either no match or more than one match after partial ordering has been considered (14.5.6.2), deduction fails and, in the declaration cases, the program is ill-formed.

#### 14.8.3 Overload resolution

[temp.over]

A function template can be overloaded either by (non-template) functions of its name or by (other) function templates of the same name. When a call to that name is written (explicitly, or implicitly using the operator notation), template argument deduction (14.8.2) and checking of any explicit template arguments (14.3) are performed for each function template to find the template argument values (if any) that can be used with

that function template to instantiate a function template specialization that can be invoked with the call arguments. For each function template, if the argument deduction and checking succeeds, the *template-arguments* (deduced and/or explicit) are used to synthesize the declaration of a single function template specialization which is added to the candidate functions set to be used in overload resolution. If, for a given function template, argument deduction fails, no such function is added to the set of candidate functions for that template. The complete set of candidate functions includes all the synthesized declarations and all of the non-template overloaded functions of the same name. The synthesized declarations are treated like any other functions in the remainder of overload resolution, except as explicitly noted in 13.3.3.<sup>145</sup>

[Example:

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.

<sup>3</sup> Here is an example involving conversions on a function argument involved in template-argument deduction:

<sup>4</sup> Here is an example involving conversions on a function argument not involved in *template-parameter* deduction:

— 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:

§ 14.8.3 375

<sup>145)</sup> 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.

```
template<class T> void f(T);  // declaration

void g() {
   f("Annemarie");  // call of f<const char*>
}
```

The call of f is well-formed even if the template f is only declared and not defined at the point of the call. The program will be ill-formed unless a specialization for f<const char\*>, either implicitly or explicitly generated, is present in some translation unit. — end example]

# 15 Exception handling

[except]

Exception handling provides a way of transferring control and information from a point in the execution of a thread to an exception handler associated with a point previously passed by the execution. A handler will be invoked only by a *throw-expression* invoked in code executed in the handler's try block or in functions called from the handler's try block.

throw  $assignment-expression_{opt}$  The optional attribute-specifier-seq in an exception-declaration appertains to the formal parameter of the catch clause (15.3).

- 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]
- <sup>3</sup> A goto or switch statement shall not be used to transfer control into a try block or into a handler. [Example:

```
void f() {
                     // Ill-formed
  goto 11;
 goto 12;
                     // Ill-formed
  try {
    goto 11;
                     //OK
                     // Ill-formed
    goto 12;
    11: ;
  } catch (...) {
    12: ;
                     // Ill-formed
    goto 11;
    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)
```

Exception handling 377

```
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

<sup>4</sup> A function-try-block associates a handler-seq with the ctor-initializer, if present, and the compound-statement. An exception thrown during the execution of the compound-statement or, for constructors and destructors, during the initialization or destruction, respectively, of the class's subobjects, transfers control to a handler in a function-try-block in the same way as an exception thrown during the execution of a try-block transfers control to other handlers. [Example:

```
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
}
```

#### 15.1 Throwing an exception

— end example]

[except.throw]

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:

§ 15.1 378

```
try {
    f(1.2);
} catch(Overflow& oo) {
    // handle exceptions of type Overflow here
}
```

— end example]

When an exception is thrown, control is transferred to the nearest handler with a matching type (15.3); "nearest" means the handler for which the *compound-statement* or *ctor-initializer* following the **try** keyword was most recently entered by the thread of control and not yet exited.

- A throw-expression initializes a temporary object, called the exception object, the type of which is determined by removing any top-level cv-qualifiers from the static type of the operand of throw and adjusting the type from "array of T" or "function returning T" to "pointer to T" or "pointer to function returning T", respectively. The temporary is an Ivalue 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.
- The memory for the exception object is allocated in an unspecified way, except as noted in 3.7.4.1. If a handler exits by rethrowing, control is passed to another handler for the same exception. The exception object is destroyed after either the last remaining active handler for the exception exits by any means other than rethrowing, or the last object of type std::exception\_ptr (18.8.5) that refers to the exception object is destroyed, whichever is later. In the former case, the destruction occurs when the handler exits, immediately after the destruction of the object declared in the exception-declaration in the handler, if any. In the latter case, the destruction occurs before the destructor of std::exception\_ptr returns. The implementation may then deallocate the memory for the exception object; any such deallocation is done in an unspecified way. [Note: an exception thrown by a throw-expression does not propagate to other threads unless caught, stored, and rethrown using appropriate library functions; see 18.8.5 and 30.6. —end note]
- 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).
- An exception is considered caught when a handler for that exception becomes active (15.3). [Note: An exception can have active handlers and still be considered uncaught if it is rethrown. —end note]
- 7 If the exception handling mechanism, after completing evaluation of the expression to be thrown but before the exception is caught, calls a function that exits via an exception, std::terminate is called (15.5.1).
  [Example:

— end example]

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:

§ 15.1 379

— 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 of any storage duration whose initialization or destruction is terminated by an exception will have destructors executed for all of its fully constructed subobjects (excluding the variant members of a union-like class), 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." If a destructor called during stack unwinding exits with an exception, std::terminate is called (15.5.1). [Note: 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, an abstract class type, or an rvalue reference type. The exception-declaration shall not denote a pointer or reference to an incomplete type, other than void\*, const void\*, volatile void\*, or const volatile void\*.
- A handler of type "array of T" or "function returning T" is adjusted to be of type "pointer to T" or "pointer to function returning T", respectively.
- A handler is a match for an exception object of type E if
  - 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:

§ 15.3 380

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]

- <sup>4</sup> 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.
- <sup>5</sup> 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.
- <sup>6</sup> If no match is found among the handlers for a try block, the search for a matching handler continues in a dynamically surrounding try block of the same thread.
- <sup>7</sup> 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.
- <sup>8</sup> The exception with the most recently activated handler that is still active is called the *currently handled* exception.
- <sup>9</sup> 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).
- Referring to any non-static member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object results in undefined behavior.
- The fully constructed base classes and members of an object shall be destroyed before entering the handler of a function-try-block of a constructor 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).
- The scope and lifetime of the parameters of a function or constructor extend into the handlers of a functiontry-block.
- Exceptions thrown in destructors of objects with static storage duration or in constructors of namespacescope 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.
- 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

§ 15.3 381

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).

- If the exception-declaration specifies a name, it declares a variable which is copy-initialized (8.5) from the exception object. If the exception-declaration denotes an object type but does not specify a name, a temporary (12.2) is copy-initialized (8.5) from the exception object. The lifetime of the variable or temporary ends when the handler exits, after the destruction of any automatic objects initialized within the handler.
- 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

[except.spec]

A function declaration lists exceptions that its function might directly or indirectly throw by using an exception-specification as a suffix of its declarator.

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).

<sup>2</sup> 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:

- 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, or
  - both are *dynamic-exception-specifications* that have the same set of adjusted types.
- <sup>4</sup> 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,

§ 15.4 382

or pointer to member function has an 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:

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:

- 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.
- 7 An exception-specification can include the same type more than once and can include classes that are related by inheritance, even though doing so is redundant. [Note: An exception-specification can also include the class std::bad\_exception (18.8.2). —end note]
- <sup>8</sup> A function is said to *allow* an exception of type E if the *constant-expression* in its *noexcept-specification* evaluates to false or 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:
class X { };
class Y { };
class Z: public X { };
class W { };
```

§ 15.4 383

[Note: A function can have multiple declarations with different non-throwing exception-specifications; for this purpose, the one on the function definition is used. -end note]

- The function 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:

— end example]

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);
};
```

§ 15.4 384

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*]

- A deallocation function (3.7.4.2) with no explicit *exception-specification* is treated as if it were specified with noexcept(true).
- <sup>16</sup> In a dynamic-exception-specification, a type-id followed by an ellipsis is a pack expansion (14.5.3).
- 17 [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 some situations exception handling must be abandoned for less subtle error handling techniques. [Note: These situations are:
  - when the exception handling mechanism, after completing the initialization of the exception object but before activation of a handler for the exception (15.1), calls a function that exits via an exception, or
  - when the exception handling mechanism cannot find a handler for a thrown exception (15.3), or
  - when the search for a handler (15.3) encounters the outermost block of a function with a noexcept-specification that does not allow the exception (15.4), or
  - when the destruction of an object during stack unwinding (15.2) terminates by throwing an exception, or
  - when initialization of a non-local variable with static or thread storage duration (3.6.2) exits via an exception, or
  - when destruction of an object with static or thread storage duration exits via an exception (3.6.3), or
  - when execution of a function registered with std::atexit or std::at\_quick\_exit exits via an exception (18.5), or
  - when a *throw-expression* 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 dynamic-exception-specification, and std::bad\_exception is not included in that dynamic-exception-specification (15.5.2), or
  - when the implementation's default unexpected exception handler is called (D.11.1), or
  - when the function std::nested\_exception::rethrow\_nested is called for an object that has captured no exception (18.8.6), or
  - when execution of the initial function of a thread exits via an exception (30.3.1.2), or
  - when the destructor or the copy assignment operator is invoked on an object of type std::thread that refers to a joinable thread (30.3.1.3, 30.3.1.4).

§ 15.5.1 385

- end note]

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, unwound partially, or not unwound at all before std::terminate() is called. In all other situations, the stack shall not be unwound before std::terminate() is called. An implementation is not permitted to finish stack unwinding prematurely based on a determination that the unwind process will eventually cause a call to std::terminate().

## 15.5.2 The std::unexpected() function

## [except.unexpected]

- <sup>1</sup> If a function with a *dynamic-exception-specification* throws an exception that is not listed in the *dynamic-exception-specification*, the function std::unexpected() is called (D.11) immediately after completing the stack unwinding for the former function.
- <sup>2</sup> [Note: By default, std::unexpected() calls std::terminate(), but a program can install its own handler function (D.11.2). In either case, the constraints in the following paragraph apply. —end note]
- 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 dynamic-exception-specification does not allow then the following happens: If the dynamic-exception-specification does not include the class std::bad\_exception (18.8.2) 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 dynamic-exception-specification was violated.
- Thus, a dynamic-exception-specification guarantees that only the listed exceptions will be thrown. If the dynamic-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

## [except.uncaught]

The function std::uncaught\_exception() returns true after completing the initialization of the exception object (15.1) until completing the activation of a handler for the exception (15.3, 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.

§ 15.5.3 386

# 16 Preprocessing directives

[cpp]

A preprocessing directive consists of a sequence of preprocessing tokens that satisfies the following constraints: The first token in the sequence is a # preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character. The last token in the sequence is the first new-line character that follows the first token in the sequence. A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

```
preprocessing-file:
       group_{opt}
group:
       group\text{-}part
       group group-part
group-part:
       if-section
       control\text{-}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<sub>opt</sub>
{\it elif-groups:}
       elif-group
       elif-groups elif-group
{\it elif-group:}
       # elif
                           constant-expression new-line group_{opt}
else\hbox{-}group:
       # else
                           new-line group_{opt}
endif-line:
       # endif
                           new-line
control\mbox{-}line:
       # include
                           pp-tokens new-line
       # define
                           identifier\ replacement-list new-line
       # define
                           identifier\ lparen\ identifier\ list_{opt}) replacement\ list\ new\ line
       # define
                           identifier\ lparen\ \dots ) replacement-list\ new-line
                           identifier\ lparen\ identifier\ list,\ \dots ) replacement\ list\ new\ line
       # define
                           identifier\ new-line
       # undef
                           pp-tokens new-line
       # line
                           pp-tokensopt new-line
         pragma
                           pp\text{-}tokens_{opt} new\text{-}line
       # new-line
text-line:
       pp-tokens<sub>opt</sub> new-line
```

<sup>146)</sup> 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).

```
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<sub>opt</sub>
pp-tokens:
      preprocessing	ext{-}token
      pp-tokens preprocessing-token
new-line:
```

the new-line character

A text line shall not begin with a # preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.

- When in a group that is skipped (16.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.
- The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).
- The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.
- The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

```
[ Example: In:
#define EMPTY
EMPTY
        #
            include <file.h>
```

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]

#### Conditional inclusion

[cpp.cond]

The expression that controls conditional inclusion shall be an integral constant expression except that identifiers (including those lexically identical to keywords) are interpreted as described below 147 and it may contain unary operator expressions of the form

```
defined identifier
```

or

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 constant-expression new-line group<sub>ont</sub> # elif constant-expression new-line group<sub>opt</sub>

§ 16.1 388

<sup>147)</sup> 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.

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<sup>148</sup>, 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. <sup>150</sup> 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.
- <sup>5</sup> Preprocessing directives of the forms
  - # ifdef  $identifier\ new-line\ group_{opt}$
  - # ifndef identifier new-line group\_ont

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.<sup>151</sup>

#### 16.2 Source file inclusion

[cpp.include]

- A #include directive shall identify a header or source file that can be processed by the implementation.
- <sup>2</sup> A preprocessing directive of the form
  - # include < h-char-sequence> new-line

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

- 3 A preprocessing directive of the form
  - # include " q-char-sequence" new-line

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

§ 16.2 389

<sup>148)</sup> 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.

<sup>149)</sup> Thus on an implementation where std::numeric\_limits<int>::max() is 0x7FFF and std::numeric\_limits<unsigned int>::max() is 0xFFFF, the integer literal 0x8000 is signed and positive within a #if expression even though it is unsigned in translation phase 7 (2.2).

<sup>150)</sup> 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.

<sup>151)</sup> 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.

# include < h-char-sequence> new-line

with the identical contained sequence (including > characters, if any) from the original directive.

4 A preprocessing directive of the form

# include pp-tokens new-line

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after include in the directive are processed just as in normal text (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined. The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of "characters is combined into a single header name preprocessing token is implementation-defined.

- <sup>5</sup> 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.
- <sup>6</sup> A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file, up to an implementation-defined nesting limit.
- 7 [Note: Although an implementation may provide a mechanism for making arbitrary source files available to the < > search, in general programmers should use the < > form for headers provided with the implementation, and the " " form for sources outside the control of the implementation. For instance:

```
#include <stdio.h>
   #include <unistd.h>
   #include "usefullib.h"
   #include "myprog.h"
     -end note
[Example: This illustrates macro-replaced #include directives:
   #if VERSION == 1
       #define INCFILE
                        "vers1.h"
   #elif VERSION == 2
       #define INCFILE
                         "vers2.h"
                                     // and so on
   #else
       #define INCFILE
                        "versN.h"
   #endif
   #include INCFILE
 — end example]
```

#### 16.3 Macro replacement

[cpp.replace]

- <sup>1</sup> Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.
- An identifier currently defined as an *object-like* macro may be redefined by another #define preprocessing directive provided that the second definition is an object-like macro definition and the two replacement lists are identical, otherwise the program is ill-formed. Likewise, an identifier currently defined as a *function-like* macro may be redefined by another #define preprocessing directive provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical, otherwise the program is ill-formed.
- 3 There shall be white-space between the identifier and the replacement list in the definition of an object-like macro.
- <sup>4</sup> 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

§ 16.3

<sup>152)</sup> 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.

than there are parameters in the macro definition (excluding the ...). There shall exist a ) preprocessing token that terminates the invocation.

- <sup>5</sup> The identifier \_\_VA\_ARGS \_\_ shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.
- <sup>6</sup> A parameter identifier in a function-like macro shall be uniquely declared within its scope.
- <sup>7</sup> 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.
- <sup>8</sup> If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.
- 9 A preprocessing directive of the form
  - # define identifier replacement-list new-line

defines an *object-like macro* that causes each subsequent instance of the macro name  $^{153}$  to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.  $^{154}$  The replacement list is then rescanned for more macro names as specified below.

- A preprocessing directive of the form
  - # define identifier lparen identifier-listopt) replacement-list new-line
  - # define  $identifier\ lparen\ \dots\ )$   $replacement\ list\ new\ line$
  - # define identifier lparen identifier-list , ... ) replacement-list new-line

defines a function-like macro with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the #define preprocessing directive. Each subsequent instance of the function-like macro name followed by a ( as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching ) preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

- The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, <sup>155</sup> the behavior is undefined.
- 12 If there is a ... in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the variable arguments. The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the ...).

### 16.3.1 Argument substitution

|cpp.subst|

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. 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.

§ 16.3.1 391

<sup>153)</sup> Since, by macro-replacement time, all character literals and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 2.2, translation phases), they are never scanned for macro names or parameters.

<sup>154)</sup> An alternative token (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.

<sup>155)</sup> Despite the name, a non-directive is a preprocessing directive.

<sup>2</sup> 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

[cpp.stringize]

<sup>1</sup> Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

A character string literal is a string-literal with no prefix. If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token comprising the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character literals: a \ character is inserted before each " and \ character of a character literal or string literal (including the delimiting " characters). If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty argument is "". The order of evaluation of # and ## operators is unspecified.

## 16.3.3 The ## operator

[cpp.concat]

- <sup>1</sup> A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.
- <sup>2</sup> 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.<sup>156</sup>
- 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:

§ 16.3.3 392

<sup>156)</sup> Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

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]

- 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.
- <sup>2</sup> If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.
- The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 16.9 below.

#### 16.3.5 Scope of macro definitions

[cpp.scope]

- 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.
- <sup>2</sup> 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.

<sup>3</sup> [Note: The simplest use of this facility is to define a "manifest constant," as in

```
#define TABSIZE 100
int table[TABSIZE];
```

- end note]

<sup>4</sup> 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.

<sup>5</sup> To illustrate the rules for redefinition and reexamination, the sequence

```
#define x
#define f(a)
                 f(x * (a))
#undef x
                 2
#define x
#define g
                 f
#define z
                 z[0]
#define h
                 g(\sim
#define m(a)
                 a(w)
#define w
                 0,1
#define t(a)
#define p()
                 int
#define q(x)
#define r(x,y)
                x ## y
#define str(x)
```

§ 16.3.5

```
f(y+1) + f(f(z)) \% t(t(g)(0) + t)(1);
     g(x+(3,4)-w) \mid h = 5 % m
         (f)^m(m);
     p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
     char c[2][6] = { str(hello), str() };
      results in
     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", "" };
<sup>6</sup> 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') // this goes away
         == 0) str(: @\n), s);
     #include xstr(INCFILE(2).h)
     glue(HIGH, LOW);
     xglue(HIGH, LOW)
      results in
     printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
     fputs("strncmp(\"abc\\0d\", \"abc\", ^{\prime}\4^{\prime}) == 0" ": ^{\circ}\n", 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\\", ^{\prime}\4^{\prime}) == 0: @\n", s);
     #include "vers2.h"
                             (after macro replacement, before file access)
     "hello";
     "hello, world"
      Space around the # and ## tokens in the macro definition is optional.
<sup>7</sup> To illustrate the rules for placemarker preprocessing tokens, the sequence
     #define t(x,y,z) x ## y ## z
     int j[] = \{ t(1,2,3), t(,4,5), t(6,,7), t(8,9,), \}
       t(10,,), t(,11,), t(,,12), t(,,) };
      results in
     int j[] = \{ 123, 45, 67, 89, \}
       10, 11, 12, };
   § 16.3.5
```

394

<sup>8</sup> To demonstrate the redefinition rules, the following sequence is valid.

But the following redefinitions are invalid:

```
#define OBJ_LIKE (0)  // different token sequence
#define OBJ_LIKE (1 - 1)  // different white space
#define FUNC_LIKE(b) ( a )  // different parameter usage
#define FUNC_LIKE(b) ( b )  // different parameter spelling
```

<sup>9</sup> Finally, to show the variable argument list macro facilities:

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test) ? puts(#test) : printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);

results in

fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y) ? puts("x>y") : printf("x is %d but y is %d", x, y));

— end note]
```

16.4 Line control

The string literal of a #line directive, if present, shall be a character string literal.

<sup>2</sup> 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.

[cpp.line]

- <sup>3</sup> A preprocessing directive of the form
  - # line digit-sequence new-line

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). If the digit sequence specifies zero or a number greater than 2147483647, the behavior is undefined.

- 4 A preprocessing directive of the form
  - # line digit-sequence "  $s\text{-}char\text{-}sequence_{opt}$ " new-line

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

- 5 A preprocessing directive of the form
  - # line pp-tokens new-line

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after line on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.

16.5 Error directive [cpp.error]

A preprocessing directive of the form

§ 16.5

# error pp-tokens<sub>opt</sub> new-line

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens, and renders the program ill-formed.

## 16.6 Pragma directive

[cpp.pragma]

- A preprocessing directive of the form
  - #  $pragma pp-tokens_{opt} new-line$

causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any pragma that is not recognized by the implementation is ignored.

#### 16.7 Null directive

[cpp.null]

- A preprocessing directive of the form
  - # new-line

has no effect.

#### 16.8 Predefined macro names

[cpp.predefined]

- The following macro names shall be defined by the implementation:
  - \_\_cplusplus

The name \_\_cplusplus is defined to the value 201103L when compiling a C++ translation unit. 157

\_ \_ 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). <sup>158</sup>

\_\_LINE\_\_

The presumed line number (within the current source file) of the current source line (an integer constant).  $^{158}$ 

\_\_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.

- <sup>2</sup> 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.

§ 16.8 396

<sup>157)</sup> 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.

<sup>158)</sup> The presumed source file name and line number can be changed by the #line directive.

```
__STDC_VERSION__ is predefined and if so, what its value is, are implementation-defined.
```

```
__STDC_ISO_10646__
```

An integer constant of the form yyyymmL (for example, 199712L). 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.

```
__STDCPP_STRICT_POINTER_SAFETY__
```

Defined, and has the value integer constant 1, if and only if the implementation has strict pointer safety (3.7.4.3).

```
__STDCPP_THREADS__
```

Defined, and has the value integer constant 1, if and only if a program can have more than one thread of execution (1.10).

- <sup>3</sup> The values of the predefined macros (except for \_\_FILE\_\_ and \_\_LINE\_\_) remain constant throughout the translation unit.
- If any of the pre-defined macro names in this subclause, or the identifier defined, is the subject of a #define or a #undef preprocessing directive, the behavior is undefined. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

### 16.9 Pragma operator

[cpp.pragma.op]

A unary operator expression of the form:

```
_Pragma ( string\text{-}literal )
```

is processed as follows: The string literal is destringized by deleting the L prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence  $\$  by a double-quote, and replacing each escape sequence  $\$  by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the pp-tokens in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

[Example:

```
#pragma listing on "..\listing.dir"
  can also be expressed as:
_Pragma ( "listing on \"..\\listing.dir\"" )
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING( ..\listing.dir )

— end example]
```

§ 16.9 397

# 17 Library introduction

[library]

17.1 General [library.general]

This Clause describes the contents of the C++ standard library, how a well-formed C++ program makes use of the library, and how a conforming implementation may provide the entities in the library.

- <sup>2</sup> 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.
- 3 Detailed specifications for each of the components in the library are in Clauses 18-30, as shown in Table 13.

Clause	Category
18	Language support library
19	Diagnostics library
20	General utilities library
21	Strings library
22	Localization library
23	Containers library
24	Iterators library
25	Algorithms library
26	Numerics library
27	Input/output library
28	Regular expressions library
29	Atomic operations library
30	Thread support library

Table 13 — Library categories

- <sup>4</sup> 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).
- 5 The diagnostics library (Clause 19) provides a consistent framework for reporting errors in a C++ program, including predefined exception classes.
- <sup>6</sup> The general utilities library (Clause 20) includes components used by other library elements, such as a predefined storage allocator for dynamic storage management (3.7.4), and components used as infrastructure in C++ programs, such as a tuples, function wrappers, and time facilities.
- <sup>7</sup> The strings library (Clause 21) provides support for manipulating text represented as sequences of type char, sequences of type char16\_t, sequences of type char32\_t, sequences of type wchar\_t, and sequences of any other character-like type.
- <sup>8</sup> The localization library (Clause 22) provides extended internationalization support for text processing.
- <sup>9</sup> The containers (Clause 23), iterators (Clause 24), and algorithms (Clause 25) libraries provide a C++ program with access to a subset of the most widely used algorithms and data structures.
- The numerics library (Clause 26) provides numeric algorithms and complex number components that extend support for numeric processing. The valarray component provides support for *n*-at-a-time processing, potentially implemented as parallel operations on platforms that support such processing. The random number component provides facilities for generating pseudo-random numbers.
- 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,

§ 17.1 398

locales, and iterators.

- <sup>12</sup> The regular expressions library (Clause 28) provides regular expression matching and searching.
- The atomic operations library (Clause 29) allows more fine-grained concurrent access to shared data than is possible with locks.

<sup>14</sup> 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

[library.c]

- <sup>1</sup> The C++ standard library also makes available the facilities of the C standard library, suitably adjusted to ensure static type safety.
- <sup>2</sup> 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 [definitions]

# 17.3.1

[defns.arbitrary.stream]

## arbitrary-positional stream

a stream (described in Clause 27) that can seek to any integral position within the length of the stream [Note: Every arbitrary-positional stream is also a repositional stream.  $-end\ note$ ]

[defns.block]

#### block

place a thread in the blocked state

17.3.3 [defns.blocked]

#### 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  $^{159}$ 

17.3.4 [defns.character]

#### character

<Clauses 21, 22, 27, and 28> any object which, when treated sequentially, can represent text
[Note: The term does not mean only char, char16\_t, char32\_t, and wchar\_t objects, but any value that can be represented by a type that provides the definitions specified in these Clauses. — end note

# 17.3.5 [defns.character.container]

#### character container type

a class or a type used to represent a *character* 

[Note: It is used for one of the template parameters of the string, iostream, and regular expression class templates. A character container type is a POD (3.9) type. — end note]

17.3.6 [defns.comparison]

#### comparison function

an operator function (13.5) for any of the equality (5.10) or relational (5.9) operators

17.3.7 [defns.component]

#### component

a group of library entities directly related as members, parameters, or return types

159) This definition is taken from POSIX.

§ 17.3

[ Note: For example, the class template basic\_string and the non-member function templates that operate on strings are referred to as the string component. — end note ]

[defns.deadlock]

#### deadlock

one or more threads are unable to continue execution because each is blocked waiting for one or more of the others to satisfy some condition

17.3.9 default behavior

[defns.default.behavior.impl]

<implementation> any specific behavior provided by the implementation, within the scope of the required behavior

17.3.10 [defns.default.behavior.func]

default behavior

<specification> a description of replacement function and handler function semantics

17.3.11 [defns.handler]

handler function

a non-reserved function whose definition may be provided by a C++ program

[ Note: A C++ program may designate a handler function at various points in its execution by supplying a pointer to the function when calling any of the library functions that install handler functions (Clause 18). — end note ]

[defns.iostream.templates]

iostream class templates

templates, defined in Clause 27, that take two template arguments

[Note: The arguments are named charT and traits. The argument charT is a character container class, and the argument traits is a class which defines additional characteristics and functions of the character type represented by charT necessary to implement the iostream class templates. — end note]

17.3.13 [defns.modifier]

modifier function

a class member function (9.3) other than a constructor, assignment operator, or destructor that alters the state of an object of the class

17.3.14 [defns.move.constr]

move construction

direct-initialization of an object of some type with an rvalue of the same type

17.3.15 [defns.move.assign]

move assignment

assignment of an rvalue of some object type to a modifiable lvalue of the same type

[defns.obj.state]

object state

the current value of all non-static class members of an object (9.2)

[Note: The state of an object can be obtained by using one or more observer functions. — end note]

[defns.ntcts]

§ 17.3 400

#### **NTCTS**

a sequence of values that have *character type* that precede the terminating null character type value <code>charT()</code>

17.3.18 [defns.observer]

#### observer function

a class member function (9.3) that accesses the state of an object of the class but does not alter that state [Note: Observer functions are specified as const member functions (9.3.2). — end note]

17.3.19 [defns.replacement]

## replacement function

a non-reserved function whose definition is provided by a C++ program

[Note: Only one definition for such a function is in effect for the duration of the program's execution, as the result of creating the program (2.2) and resolving the definitions of all translation units (3.5). — end note]

17.3.20 [defns.repositional.stream]

#### repositional stream

a stream (described in Clause 27) that can seek to a position that was previously encountered

(.... )

# 17.3.21 [defns.required.behavior]

#### required behavior

a description of *replacement function* and *handler function* semantics applicable to both the behavior provided by the implementation and the behavior of any such function definition in the program

[Note: If such a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined. -end note]

17.3.22 [defns.reserved.function]

#### reserved function

a function, specified as part of the C++ standard library, that must be defined by the implementation [Note: If a C++ program provides a definition for any reserved function, the results are undefined. — end note]

17.3.23 [defns.stable]

#### stable algorithm

an algorithm that preserves, as appropriate to the particular algorithm, the order of elements [Note: Requirements for stable algorithms are given in 17.6.5.7. — end note]

17.3.24 [defns.traits]

#### traits class

a class that encapsulates a set of types and functions necessary for class templates and function templates to manipulate objects of types for which they are instantiated

[Note: Traits classes defined in Clauses 21, 22 and 27 are character traits, which provide the character handling support needed by the string and iostream classes. — end note]

17.3.25 [defns.unblock]

#### unblock

place a thread in the unblocked state

17.3.26 [defns.valid]

#### valid but unspecified state

an object state that is not specified except that the object's invariants are met and operations on the object

§ 17.3 401

behave as specified for its type

[Example: If an object x of type std::vector<int> is in a valid but unspecified state, x.empty() can be called unconditionally, and x.front() can be called only if x.empty() returns false. — end example]

#### 17.4 Additional definitions

[defns.additional]

1.3 defines additional terms used elsewhere in this International Standard.

### 17.5 Method of description (Informative)

[description]

This subclause describes the conventions used to specify the C++ standard library. 17.5.1 describes the structure of the normative Clauses 18 through 30 and Annex D. 17.5.2 describes other editorial conventions.

#### 17.5.1 Structure of each clause

[structure]

17.5.1.1 Elements

[structure.elements]

- Each library clause contains the following elements, as applicable: 160
  - Summary
  - Requirements
  - Detailed specifications
  - References to the Standard C library

## 17.5.1.2 Summary

[structure.summary]

- <sup>1</sup> 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.
- <sup>2</sup> Paragraphs labeled "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

#### 17.5.1.3 Requirements

[structure.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
- <sup>2</sup> 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.
- <sup>3</sup> 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.

§ 17.5.1.3 402

<sup>160)</sup> 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.

<sup>4</sup> 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.<sup>161</sup>

## 17.5.1.4 Detailed specifications

[structure.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
- <sup>2</sup> Descriptions of class member functions follow the order (as appropriate): <sup>162</sup>
  - constructor(s) and destructor
  - copying, moving & assignment functions
  - comparison functions
  - modifier functions
  - observer functions
  - operators and other non-member functions
- <sup>3</sup> Descriptions of function semantics contain the following elements (as appropriate): <sup>163</sup>
  - 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

§ 17.5.1.4 403

<sup>161)</sup> Although in some cases the code given is unambiguously the optimum implementation.

<sup>162)</sup> 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.

<sup>163)</sup> 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.

- 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.
- <sup>5</sup> 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.
- 6 If the formulation of a complexity requirement calls for a negative number of operations, the actual requirement is zero operations. 164
- Complexity requirements specified in the library clauses are upper bounds, and implementations that provide better complexity guarantees satisfy the requirements.
- <sup>8</sup> 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).

17.5.1.5 C library [structure.see.also]

Paragraphs labeled "See also:" contain cross-references to the relevant portions of this International Standard and the ISO C standard, which is incorporated into this International Standard by reference.

#### 17.5.2 Other conventions

[conventions]

This subclause describes several editorial conventions used to describe the contents of the C++ standard library. These conventions are for describing implementation-defined types (17.5.2.1), and member functions (17.5.2.2).

## 17.5.2.1 Type descriptions

[type.descriptions]

#### 17.5.2.1.1 General

[type.descriptions.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.
- <sup>2</sup> 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

[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. 166
- <sup>2</sup> The enumerated type enumerated can be written:

```
enum enumerated { VO , V1 , V2 , V3 , ..... };
static const enumerated CO (VO);
```

§ 17.5.2.1.2

<sup>164)</sup> This simplifies the presentation of complexity requirements in some cases.

<sup>165)</sup> Examples from 17.6.3 include: EqualityComparable, LessThanComparable, CopyConstructible. Examples from 24.2 include: InputIterator, ForwardIterator, Function, Predicate.

<sup>166)</sup> Such as an integer type, with constant integer values (3.9.1).

```
static const enumerated C1 (V1);
static const enumerated C2 (V2);
static const enumerated C3 (V3);
```

<sup>3</sup> 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

[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).
- <sup>2</sup> The bitmask type *bitmask* can be written:

```
// For exposition only.
// int_type is an integral type capable of
// representing all values of the bitmask type.
enum bitmask : int_type {
  VO = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, ....
};
constexpr bitmask CO (VO);
constexpr bitmask C1 (V1);
constexpr bitmask C2 (V2);
constexpr bitmask C3 (V3);
  . . . . .
constexpr bitmask operator&(bitmask X, bitmask Y) {
  return static_cast<br/>bitmask>(
    static_cast<int_type>(X) & static_cast<int_type>(Y));
constexpr bitmask operator|(bitmask X, bitmask Y) {
 return static_cast<bitmask>(
    static_cast<int_type>(X) | static_cast<int_type>(Y));
constexpr bitmask operator^(bitmask X, bitmask Y){
  return static_cast<br/>bitmask>(
    static_cast<int_type>(X) ^ static_cast<int_type>(Y));
constexpr bitmask operator~(bitmask X){
  return static_cast<bitmask>(~static_cast<int_type>(X));
}
bitmask& operator&=(bitmask& X, bitmask Y){
  X = X & Y; return X;
}
bitmask& operator | = (bitmask& X, bitmask Y) {
  X = X \mid Y; return X;
bitmask& operator^=(bitmask& X, bitmask Y) {
  X = X ^ Y; return X;
```

- <sup>3</sup> Here, the names C0, C1, etc. represent *bitmask elements* for this particular bitmask type. All such elements have distinct values such that, for any pair Ci and Cj, Ci & Ci is nonzero and Ci & Cj is zero.
- <sup>4</sup> 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.

§ 17.5.2.1.3 405

- 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

[character.seq]

- 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. 167
  - The decimal-point character is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in Clauses 18 through 30 and Annex D by a period, '.', which is also its value in the "C" locale, but may change during program execution by a call to setlocale(int, const char\*), 168 or by a change to a locale object, as described in Clauses 22.3 and 27.
  - 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 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

[byte.strings]

- <sup>1</sup> 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. <sup>169</sup>
- <sup>2</sup> The *length* of an NTBS is the number of elements that precede the terminating null character. An *empty* NTBS has a length of zero.
- <sup>3</sup> The *value* of an NTBS is the sequence of values of the elements up to and including the terminating null character.
- <sup>4</sup> A static NTBS is an NTBS with static storage duration. <sup>170</sup>

## 17.5.2.1.4.2 Multibyte strings

[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.<sup>171</sup>
- <sup>2</sup> A static NTMBS is an NTMBS with static storage duration.

#### 17.5.2.2 Functions within classes

[functions.within.classes]

- <sup>1</sup> 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).
- <sup>2</sup> It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

## 17.5.2.3 Private members

[objects.within.classes]

<sup>1</sup> 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

§ 17.5.2.3

<sup>167)</sup> Note that this definition differs from the definition in ISO C 7.1.1.

<sup>168)</sup> declared in <clocale> (22.6).

<sup>169)</sup> 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. 170) A string literal, such as "abc", is a static NTBS.

<sup>171)</sup> 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.

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 exposition only, as in:

streambuf\* sb; // exposition only

3 An implementation may use any technique that provides equivalent external behavior.

## 17.6 Library-wide requirements

[requirements]

- This subclause specifies requirements that apply to the entire C++ standard library. Clauses 18 through 30 and Annex D specify the requirements of individual entities within the library.
- Requirements specified in terms of interactions between threads do not apply to programs having only a single thread of execution.
- <sup>3</sup> Within this subclause, 17.6.1 describes the library's contents and organization, 17.6.2 describes how well-formed C++ programs gain access to library entities, 17.6.3 describes constraints on types and functions used with the C++ standard library, 17.6.4 describes constraints on well-formed C++ programs, and 17.6.5 describes constraints on conforming implementations.

#### 17.6.1 Library contents and organization

[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

[contents]

- The C++ standard library provides definitions for the following types of entities: macros, values, types, templates, classes, functions, objects.
- All library entities except macros, operator new and operator delete are defined within the namespace std or namespaces nested within namespace std.<sup>172</sup> It is unspecified whether names declared in a specific namespace are declared directly in that namespace or in an inline namespace inside that namespace.<sup>173</sup>
- 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 [headers]

- $^{1}$  Each element of the C++ standard library is declared or defined (as appropriate) in a header.  $^{174}$
- <sup>2</sup> The C++ standard library provides 52 C++ library headers, as shown in Table 14.
- 3 The facilities of the C standard Library are provided in 26 additional headers, as shown in Table 15.
- Except as noted in Clauses 18 through 30 and Annex D, the contents of each header cname shall be the same as that of the corresponding header name.h, as specified in the C standard library (1.2) or the C Unicode TR, as appropriate, as if by inclusion. In the C++ standard library, however, the declarations (except for names which are defined as macros in C) are within namespace scope (3.3.6) of the namespace std. It is unspecified whether these names are first declared within the global namespace scope and are then injected into namespace std by explicit using-declarations (7.3.3).
- Names which are defined as macros in C shall be defined as macros in the C++ standard library, even if C grants license for implementation as functions. [Note: The names defined as macros in C include the following: assert, offsetof, setjmp, va\_arg, va\_end, and va\_start. —end note]

§ 17.6.1.2 407

<sup>172)</sup> The C standard library headers (Annex D.5) 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.

<sup>173)</sup> This gives implementers freedom to use inline namespaces to support multiple configurations of the library.

<sup>174)</sup> 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 14 — C++ library headers

<algorithm></algorithm>	<fstream></fstream>	<li>t&gt;</li>	<regex></regex>	<tuple></tuple>
<array></array>	<functional></functional>	<locale></locale>	<pre><scoped_allocator></scoped_allocator></pre>	<type_traits></type_traits>
<atomic></atomic>	<future></future>	<map></map>	<set></set>	<typeindex></typeindex>
   	<pre><initializer_list></initializer_list></pre>	<memory></memory>	<sstream></sstream>	<typeinfo></typeinfo>
<chrono></chrono>	<iomanip></iomanip>	<mutex></mutex>	<stack></stack>	<pre><unordered_map></unordered_map></pre>
<codecvt></codecvt>	<ios></ios>	<new></new>	<stdexcept></stdexcept>	<pre><unordered_set></unordered_set></pre>
<pre><complex></complex></pre>	<iosfwd></iosfwd>	<numeric></numeric>	<streambuf></streambuf>	<utility></utility>
<pre><condition_variable></condition_variable></pre>	<iostream></iostream>	<pre><ostream></ostream></pre>	<string></string>	<valarray></valarray>
<deque></deque>	<istream></istream>	<queue></queue>	<strstream></strstream>	<vector></vector>
<exception></exception>	<iterator></iterator>	<random></random>	<system_error></system_error>	
<forward_list></forward_list>	<li>imits&gt;</li>	<ratio></ratio>	<thread></thread>	

Table 15 — C++ headers for C library facilities

<cassert></cassert>	<cinttypes></cinttypes>	<csignal></csignal>	<cstdio></cstdio>	<cwchar></cwchar>
<ccomplex></ccomplex>	<ciso646></ciso646>	<cstdalign></cstdalign>	<cstdlib></cstdlib>	<cwctype></cwctype>
<cctype></cctype>	<climits></climits>	<cstdarg></cstdarg>	<cstring></cstring>	
<cerrno></cerrno>	<clocale></clocale>	<cstdbool></cstdbool>	<ctgmath></ctgmath>	
<cfenv></cfenv>	<cmath></cmath>	<cstddef></cstddef>	<ctime></ctime>	
<cfloat></cfloat>	<csetjmp></csetjmp>	<cstdint></cstdint>	<cuchar></cuchar>	

- 6 Names that are defined as functions in C shall be defined as functions in the C++ standard library.<sup>175</sup>
- $^7$  Identifiers that are keywords or operators in C++ shall not be defined as macros in C++ standard library headers.  $^{176}$
- 8 D.5, C standard library headers, describes the effects of using the name.h (C header) form in a C++ program.<sup>177</sup>

#### 17.6.1.3 Freestanding implementations

[compliance]

- <sup>1</sup> Two kinds of implementations are defined: *hosted* and *freestanding* (1.4). For a hosted implementation, this International Standard describes the set of available headers.
- <sup>2</sup> A freestanding implementation has an implementation-defined set of headers. This set shall include at least the headers shown in Table 16.
- <sup>3</sup> The supplied version of the header <cstdlib> shall declare at least the functions abort, atexit, at\_quick\_exit, exit, and quick\_exit (18.5). The other headers listed in this table shall meet the same requirements as for a hosted implementation.

# 17.6.2 Using the library

[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

§ 17.6.2.1 408

<sup>175)</sup> 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.

<sup>176)</sup> In particular, including the standard header <iso646.h> or <ciso646> has no effect.

<sup>177)</sup> 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.

	Subclause	Header(s)
		<ciso646></ciso646>
18.2	Types	<cstddef></cstddef>
18.3	Implementation properties	<cfloat> <limits> <climits></climits></limits></cfloat>
18.4	Integer types	<cstdint></cstdint>
18.5	Start and termination	<cstdlib></cstdlib>
18.6	Dynamic memory management	<new></new>
18.7	Type identification	<typeinfo></typeinfo>
18.8	Exception handling	<exception></exception>
18.9	Initializer lists	<pre><initializer_list></initializer_list></pre>
18.10	Other runtime support	<cstdalign> <cstdarg> <cstdbool></cstdbool></cstdarg></cstdalign>
20.9	Type traits	<type_traits></type_traits>
29	Atomics	<atomic></atomic>

Table 16 — C++ headers for freestanding implementations

describes effects during translation phase 4, while 17.6.2.3 describes effects during phase 8 (2.2).

17.6.2.2 Headers [using.headers]

- The entities in the C++ standard library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate #include preprocessing directive (16.2).
- <sup>2</sup> 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.<sup>178</sup>
- 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 [using.linkage]

- <sup>1</sup> 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).
- Whether a name from the C standard library declared with external linkage has extern "C" or extern "C++" linkage is implementation-defined. It is recommended that an implementation use extern "C++" linkage for this purpose.<sup>179</sup>
- <sup>3</sup> Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

SEE ALSO: replacement functions (17.6.4.6), run-time changes (17.6.4.7).

#### 17.6.3 Requirements on types and expressions

[utility.requirements]

<sup>1</sup> 17.6.3.1 describes requirements on types and expressions used to instantiate templates defined in the C++ standard library. 17.6.3.2 describes the requirements on swappable types and swappable expressions. 17.6.3.3 describes the requirements on pointer-like types that support null values. 17.6.3.4 describes the requirements on hash function objects. 17.6.3.5 describes the requirements on storage allocators.

#### 17.6.3.1 Template argument requirements

[utility.arg.requirements]

The template definitions in the C++ standard library refer to various named requirements whose details are set out in tables 17–24. In these tables, T is an object or reference type to be supplied by a C++ program instantiating a template; a, b, and c are values of type (possibly const) T; s and t are modifiable lvalues of

<sup>178)</sup> This is the same as the Standard C library.

<sup>179)</sup> 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.

type T; u denotes an identifier; rv is an rvalue of type T; and v is an lvalue of type (possibly const) T or an rvalue of type const T.

In general, a default constructor is not required. Certain container class member function signatures specify T() as a default argument. T() shall be a well-defined expression (8.5) if one of those signatures is called using the default argument (8.3.6).

Table 17 — EqualityComparable requirements [equalitycomparable]

Expression	Return type	Requirement
a == b	convertible to bool	== is an equivalence relation, that is, it has the following properties:
		— For all a, a == a.
		— If a == b, then b == a.
		— If $a == b$ and $b == c$ , then $a == c$ .

Table 18 — LessThanComparable requirements [lessthancomparable]

Expression	Return type	Requirement
a < b	convertible to	< is a strict weak ordering relation (25.4)
	bool	

Table 19 — DefaultConstructible requirements [defaultconstructible]

Expression	Post-condition
T t;	object t is default-initialized
T u{};	object u is value-initialized
T()	a temporary object of type T is value-initialized
T{}	

Table 20 — MoveConstructible requirements [moveconstructible]

Expression	Post-condition Post-condition
T u = rv;	u is equivalent to the value of rv before the construction
T(rv)	T(rv) is equivalent to the value of rv before the construction
rv's state is unsp	pecified [Note:rv must still meet the requirements of the library compo-
nent that is usin	g it. The operations listed in those requirements must work as specified
whether rv has	been moved from or not. — end note]

Table 21 — CopyConstructible requirements (in addition to MoveConstructible) [copyconstructible]

Expression	Post-condition
T u = v;	the value of $v$ is unchanged and is equivalent to $u$
T(v)	the value of $v$ is unchanged and is equivalent to $T(v)$

Table 22 — MoveAssignable requirements [moveassignable]

Expression	Return type	Return value	Post-condition
t = rv	T&	t	t is equivalent to the value of
			rv before the assignment
rv's state is u	inspecified. [Note:	rv must still meet	the requirements of the library
component that is using it. The operations listed in those requirements must work as			
specified wheth	her rv has been move	ed from or not. —	$end\ note]$

Table 23 — CopyAssignable requirements (in addition to MoveAssignable) [copyassignable]

Expression	Return type	Return value	Post-condition
t = v	T&	t	t is equivalent to v, the value of
			v is unchanged

Table 24 — Destructible requirements [destructible]

Expression	Post-condition
u.~T()	All resources owned by u are reclaimed, no exception is propagated.

#### 17.6.3.2 Swappable requirements

## [swappable.requirements]

- This subclause provides definitions for swappable types and expressions. In these definitions, let t denote an expression of type T, and let u denote an expression of type U.
- An object t is swappable with an object u if and only if:
  - the expressions swap(t, u) and 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.
- The context in which swap(t, u) and swap(u, t) are evaluated shall ensure that a binary non-member function named "swap" is selected via overload resolution (13.3) on a candidate set that includes:
  - the two swap function templates defined in  $\langle \mathtt{utility} \rangle$  (20.2) and
  - the lookup set produced by argument-dependent lookup (3.4.2).

[Note: If T and U are both fundamental types or arrays of fundamental types and the declarations from the header <utility> are in scope, the overall lookup set described above is equivalent to that of the qualified name lookup applied to the expression std::swap(t, u) or std::swap(u, t) as appropriate. — end note]

[Note: It is unspecified whether a library component that has a swappable requirement includes the header  $\langle \mathtt{utility} \rangle$  to ensure an appropriate evaluation context. — end note]

- <sup>4</sup> An rvalue or lvalue t is *swappable* if and only if t is swappable with any rvalue or lvalue, respectively, of type T.
- A type X satisfying any of the iterator requirements (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 swap calls is performed in an appropriate context under the various conditions as follows:

```
#include <utility>
 // Requires: std::forward<T>(t) shall be swappable with std::forward<U>(u).
 template <class T, class U>
 void value_swap(T&& t, U&& u) {
   using std::swap;
   swap(std::forward<T>(t), std::forward<U>(u)); // OK: uses "swappable with" conditions
                                                    // for rvalues and lvalues
 }
 // Requires: lvalues of T shall be swappable.
 template <class T>
 void lv_swap(T& t1 T& t2) {
   using std::swap;
   swap(t1, t2);
                                                     // OK: uses swappable conditions for
                                                     // lvalues of type T
 namespace N {
   struct A { int m; };
   struct Proxy { A *a; };
   Proxy proxy(A& a) { return Proxy{ &a }; }
   void swap(A& x, Proxy p) {
                                                    // OK: uses context equivalent to swappable
     std::swap(x.m, p.a->m);
                                                    // conditions for fundamental types
   }
   void swap(Proxy p, A& x) { swap(x, p); }
                                                    // satisfy symmetry constraint
 int main() {
   int i = 1, j = 2;
   lv_swap(i, j);
   assert(i == 2 && j == 1);
   N::A a1 = { 5 }, a2 = { -5 };
   value_swap(a1, proxy(a2));
   assert(a1.m == -5 \&\& a2.m == 5);
— end example]
```

#### 17.6.3.3 NullablePointer requirements

[nullablepointer.requirements]

A NullablePointer type is a pointer-like type that supports null values. A type P meets the requirements

#### of NullablePointer if:

 P satisfies the requirements of EqualityComparable, DefaultConstructible, CopyConstructible, CopyAssignable, and Destructible,

- lvalues of type P are swappable (17.6.3.2),
- the expressions shown in Table 25 are valid and have the indicated semantics, and
- P satisfies all the other requirements of this subclause.
- A value-initialized object of type P produces the null value of the type. The null value shall be equivalent only to itself. A default-initialized object of type P may have an indeterminate value. [Note: Operations involving indeterminate values may cause undefined behavior. —end note]
- <sup>3</sup> 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.
- <sup>4</sup> No operation which is part of the NullablePointer requirements shall exit via an exception.
- In Table 25, u denotes an identifier, t denotes a non-const lvalue of type P, a and b denote values of type (possibly const) P, and np denotes a value of type (possibly const) std::nullptr\_t.

Expression	Return type	Operational semantics
P u(np);		post: u == nullptr
Pu = np;		
P(np)		post: P(np) == nullptr
t = np	P&	post: t == nullptr
a != b	contextually convertible to bool	!(a == b)
a == np	contextually convertible to bool	a == P()
np == a		
a != np	contextually convertible to bool	!(a == np)
np != a		

Table 25 — NullablePointer requirements [nullablepointer]

## 17.6.3.4 Hash requirements

[hash.requirements]

- A type H meets the Hash requirements if:
  - it is a function object type (20.8),
  - it satisfies the requirements of CopyConstructible and Destructible (17.6.3.1), and
  - the expressions shown in Table 26 are valid and have the indicated semantics.
- Given Key is an argument type for function objects of type H, in Table 26 h is a value of type (possibly const) H, u is an lvalue of type Key, and k is a value of a type convertible to (possibly const) Key.

#### 17.6.3.5 Allocator requirements

#### [allocator.requirements]

The library describes a standard set of requirements for *allocators*, which are class-type objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for it. All of the string types (Clause 21), containers (Clause 23) (except array), string buffers and string streams (Clause 27), and match\_results (Clause 28) are parameterized in terms of allocators.

Table 26 —	Hash	requirements	$[\mathbf{hash}]$	
------------	------	--------------	-------------------	--

Expression	Return type	Requirement
h(k)	size_t	The value returned shall depend only on the argument k. [Note: Thus all evaluations of the expression h(k) with the same value for k yield the same result. —end note] [Note: For two different values t1 and t2, the probability that h(t1) and h(t2) compare equal should be very small, approaching 1.0 / numeric_limits <size_t>::max(). —end note]</size_t>
h(u)	size_t	Shall not modify u.

The template struct allocator\_traits (20.6.8) supplies a uniform interface to all allocator types. Table 27 describes the types manipulated through allocators. Table 28 describes the requirements on allocator types and thus on types used to instantiate allocator\_traits. A requirement is optional if the last column of Table 28 specifies a default for a given expression. Within the standard library allocator\_traits template, an optional requirement that is not supplied by an allocator is replaced by the specified default expression. A user specialization of allocator\_traits may provide different defaults and may provide defaults for different requirements than the primary template. Within Tables 27 and 28, the use of move and forward always refers to std::move and std::forward, respectively.

Table 27 — Descriptive variable definitions

Variable	Definition
T, U, C	any non-const object type (3.9)
V	a type convertible to T
Х	an Allocator class for type T
Y	the corresponding Allocator class for type U
XX	the type allocator_traits <x></x>
YY	the type allocator_traits <y></y>
t	a value of type const T&
a, a1, a2	values of type X&
a3	an rvalue of type X
b	a value of type Y
С	a dereferenceable pointer of type C*
р	a value of type XX::pointer, obtained by calling
	a1.allocate, where a1 == a
q	a value of type XX::const_pointer obtained by
	conversion from a value p.
w	a value of type XX::void_pointer obtained by conversion
	from a value p
z	a value of type XX::const_void_pointer obtained by
	conversion from a value ${\tt q}$ or a value ${\tt w}$
r	a value of type T& obtained by the expression *p.
s	a value of type $const$ T& obtained by the expression $*q$ or
	by conversion from a value r.
u	a value of type YY::const_pointer obtained by calling
	YY::allocate, or else nullptr.

 $\odot$  ISO/IEC N3337

Table 27 — Descriptive variable definitions (continued)

Variable	Definition		
v	a value of type V		
n	a value of type XX::size_type.		
Args	a template parameter pack		
args	a function parameter pack with the pattern Args&&		

Table 28 — Allocator requirements

Expression	Return type	$\begin{array}{c} {\bf Assertion/note} \\ {\bf pre-/post-condition} \end{array}$	Default
X::pointer			T*
X::const_pointer		X::pointer is convertible to X::const_pointer	<pre>pointer traits<x:: pointer="">:: rebind<const t=""></const></x::></pre>
X::void_pointer		X::pointer is convertible to	pointer
Y::void_pointer		X::void_pointer.	traits <x::< td=""></x::<>
		X::void_pointer and	pointer>::
		Y::void_pointer are the same	rebind <void></void>
		type.	
X::const_void		X::pointer,	pointer
pointer		X::const_pointer, and	traits <x::< td=""></x::<>
Y::const_void		X::void_pointer are	pointer>::
pointer		convertible to	rebind <const< td=""></const<>
		X::const_void_pointer.	void>
		X::const_void_pointer and	
		Y::const_void_pointer are	
		the same type.	
X::value_type	Identical to T		
X::size_type	unsigned integer type	a type that can represent the	make
		size of the largest object in the	unsigned <x::< td=""></x::<>
		allocation model.	difference
			type>::type
<pre>X::difference_type</pre>	signed integer type	a type that can represent the	pointer
		difference between any two	traits <x::< td=""></x::<>
		pointers in the allocation	<pre>pointer&gt;::</pre>
		model.	difference
			type
typename	Y	For all U (including T),	See Note A,
X::template		Y::template	below.
rebind <u>::other</u>	T) û	rebind <t>::other is X.</t>	
*p	T&		
*q	const T&	*q refers to the same object as	
	t	*p	
p->m	type of T::m	pre: (*p).m is well-defined.	
		equivalent to (*p).m	

 $\odot$  ISO/IEC N3337

Table 28 — Allocator requirements (continued)

Expression	Return type	$egin{array}{l} {f Assertion/note} \ {f pre-/post-condition} \end{array}$	Default
G->m	type of T::m	pre: (*q).m is well-defined.	
q->m	type or 1m	equivalent to (*q).m	
static	X::pointer	static_cast <x::pointer>(w)</x::pointer>	
cast <x::pointer>(w)</x::pointer>	xpointer	== p	
static_cast <x< td=""><td>X::const_pointer</td><td>static_cast<x< td=""><td></td></x<></td></x<>	X::const_pointer	static_cast <x< td=""><td></td></x<>	
::const_pointer>(z)	nonbo_pointoi	::const_pointer>(z) == q	
a.allocate(n)	X::pointer	Memory is allocated for n	
		objects of type T but objects	
		are not constructed. allocate	
		may raise an appropriate	
		exception. $^{180}[Note: If n == 0,$	
		the return value is unspecified.	
		$-\mathit{end}\;\mathit{note}]$	
a.allocate(n, u)	X::pointer	Same as a.allocate(n). The	a.allocate(n)
		use of u is unspecified, but it is	
		intended as an aid to locality.	
a.deallocate(p,n)	(not used)	All n T objects in the area	
		pointed to by <b>p</b> 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]	
a.max_size()	X::size_type	the largest value that can	numeric
		meaningfully be passed to	limits <size< td=""></size<>
1 0	1 1	X::allocate()	type>::max()
a1 == a2	bool	returns true only if storage allocated from each can be	
		deallocated via the other.	
		operator== shall be reflexive,	
		symmetric, and transitive, and	
		shall not exit via an exception.	
a1 != a2	bool	same as ! (a1 == a2)	
a == b	bool	same as a ==	
		Y::rebind <t>::other(b)</t>	
a != b	bool	same as !(a == b)	
X a1(a);		Shall not exit via an exception.	
,		post: a1 == a	
X a(b);		Shall not exit via an exception.	
		post: $Y(a) == b, a == X(b)$	
<pre>X a1(move(a));</pre>		Shall not exit via an exception.	
		post: a1 equals the prior value	
		of a.	

Table 28 —	Allocator	requirements	(continued)

Expression	Return type	$\begin{array}{c} {\bf Assertion/note} \\ {\bf pre-/post\text{-}condition} \end{array}$	Default
X a(move(b));		Shall not exit via an exception. post: a equals the prior value of X(b).	
a.construct(c, args)	(not used)	Effect: Constructs an object of type C at c	<pre>::new ((void*)c) C(forward&lt; Args&gt; (args))</pre>
a.destroy(c)	(not used)	Effect: Destroys the object at c	c->~C()
<pre>a.select_on container_copy construction()</pre>	X	Typically returns either a or $X()$	return a;
X::propagate_on container_copy assignment	Identical to or derived from true_type or false_type	true_type only if an allocator of type X should be copied when the client container is copy-assigned.	false_type
X::propagate_on container_move assignment	Identical to or derived from true_type or false_type	true_type only if an allocator of type X should be moved when the client container is move-assigned.	false_type
X::propagate_on container_swap	Identical to or derived from true_type or false_type	true_type only if an allocator of type X should be swapped when the client container is swapped.	false_type

- 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.
- The X::pointer, X::const\_pointer, X::void\_pointer, and X::const\_void\_pointer types shall satisfy the requirements of NullablePointer (17.6.3.3). No constructor, comparison operator, copy operation, move operation, or swap operation on these types shall exit via an exception. X::pointer and X::const\_pointer shall also satisfy the requirements for a random access iterator (24.2).
- <sup>5</sup> 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 28:

#### template <class Tp>

<sup>180)</sup> 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.

```
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);
};

— end example]
```

<sup>6</sup> 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]

### 17.6.4 Constraints on programs

[constraints]

#### 17.6.4.1 Overview

[constraints.overview]

This section describes restrictions on C++ programs that use the facilities of the C++ standard library. The following subclauses specify constraints on the program's use of namespaces (17.6.4.2.1), its use of various reserved names (17.6.4.3), its use of headers (17.6.4.4), its use of standard library classes as base classes (17.6.4.5), its definitions of replacement functions (17.6.4.6), and its installation of handler functions during execution (17.6.4.7).

## 17.6.4.2 Namespace use

[namespace.constraints]

#### 17.6.4.2.1 Namespace std

[namespace.std]

- <sup>1</sup> 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.<sup>181</sup>
- 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.

<sup>3</sup> A translation unit shall not declare namespace std to be an inline namespace (7.3.1).

# 17.6.4.2.2 Namespace posix

[namespace.posix]

The behavior of a C++ program is undefined if it adds declarations or definitions to namespace posix or to a namespace within namespace posix unless otherwise specified. The namespace posix is reserved for use by ISO/IEC 9945 and other POSIX standards.

### 17.6.4.3 Reserved names

[reserved.names]

The C++ standard library reserves the following kinds of names:

— macros

§ 17.6.4.3 418

<sup>181)</sup> 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.

- global names
- names with external linkage

<sup>2</sup> 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.

#### 17.6.4.3.1 Macro names

[macro.names]

- A translation unit that includes a standard library header shall not #define or #undef names declared in any standard library header.
- A translation unit shall not #define or #undef names lexically identical to keywords, to the identifiers listed in Table 3, or to the attribute-tokens described in 7.6.

#### 17.6.4.3.2 Global names

[global.names]

- <sup>1</sup> 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.

#### 17.6.4.3.3 External linkage

[extern.names]

- 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, <sup>182</sup> both in namespace std and in the global namespace.
- <sup>2</sup> Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage. <sup>183</sup>
- Each name from the Standard C library declared with external linkage is reserved to the implementation for use as a name with extern "C" linkage, both in namespace std and in the global namespace.
- Each function signature from the Standard C library declared with external linkage is reserved to the implementation for use as a function signature with both extern "C" and extern "C++" linkage, 184 or as a name of namespace scope in the global namespace.

17.6.4.3.4 Types [extern.types]

For each type T from the Standard C library, <sup>185</sup> the types :: T and std:: T are reserved to the implementation and, when defined, :: T shall be identical to std:: T.

## 17.6.4.3.5 User-defined literal suffixes

[usrlit.suffix]

Literal suffix identifiers that do not start with an underscore are reserved for future standardization.

#### 17.6.4.4 Headers

alt.heade

<sup>1</sup> If a file with a name equivalent to the derived file name for one of the C++ standard library headers is not provided as part of the implementation, and a file with that name is placed in any of the standard places for a source file to be included (16.2), the behavior is undefined.

#### 17.6.4.5 Derived classes

[derived.classes]

<sup>1</sup> Virtual member function signatures defined for a base class in the C++ standard library may be overridden in a derived class defined in the program (10.3).

# 17.6.4.6 Replacement functions

[replacement.functions]

1 Clauses 18 through 30 and Annex D describe the behavior of numerous functions defined by the C++ standard

§ 17.6.4.6 419

<sup>182)</sup> The list of such reserved names includes errno, declared or defined in <cerrno>.

<sup>183)</sup> 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>.

<sup>184)</sup> The function signatures declared in <cuchar>, <cwchar>, and <cwctype> are always reserved, notwithstanding the restrictions imposed in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.

<sup>185)</sup> These types are clock\_t, div\_t, FILE, fpos\_t, lconv, ldiv\_t, mbstate\_t, ptrdiff\_t, sig\_atomic\_t, size\_t, time\_t, tm, va\_list, wctrans\_t, wctype\_t, and wint\_t.

library. Under some circumstances, however, certain of these function descriptions also apply to replacement functions defined in the program (17.3).

A C++ program may provide the definition for any of eight dynamic memory allocation function signatures declared in header <new> (3.7.4, 18.6):

```
- 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&)
- <sup>3</sup> 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.

#### 17.6.4.7 Handler functions

[handler.functions]

- The C++ standard library provides default versions of the following handler functions (Clause 18):
  - unexpected\_handler
  - terminate\_handler
- <sup>2</sup> 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.

- <sup>3</sup> A C++ program can get a pointer to the current handler function by calling the following functions:
  - get\_new\_handler
  - get\_unexpected
  - get\_terminate
- 4 Calling the set\_\* and get\_\* functions shall not incur a data race. A call to any of the set\_\* functions shall synchronize with subsequent calls to the same set\_\* function and to the corresponding get\_\* function.

§ 17.6.4.7 420

# 17.6.4.8 Other functions

[res.on.functions]

In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ standard library depends on components supplied by a C++ program. If these components do not meet their requirements, the Standard places no requirements on the implementation.

- <sup>2</sup> 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, D.11.1), 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 (17.6.3.5, 23.2, 24.2, 26.2). Operations on such types can report a failure by throwing an exception unless otherwise specified.
  - 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.

## 17.6.4.9 Function arguments

[res.on.arguments]

- Each of the following 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 Ivalue of type A is bound, the argument binds to an Ivalue reference (14.8.2.1) and thus is not covered by the previous sentence. —end note] [Note: If a program casts an Ivalue to an xvalue while passing that Ivalue 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 Ivalue as a temporary. The implementation is free to optimize away aliasing checks which might be needed if the argument was an Ivalue. —end note]

# 17.6.4.10 Shared objects and the library

[res.on.objects]

- <sup>1</sup> The behavior of a program is undefined if calls to standard library functions from different threads may introduce a data race. The conditions under which this may occur are specified in 17.6.5.9. [Note: Modifying an object of a standard library type that is shared between threads risks undefined behavior unless objects of that type are explicitly specified as being sharable without data races or the user supplies a locking mechanism. —end note]
- 2 [Note: In particular, the program is required to ensure that completion of the constructor of any object of a class type defined in the standard library happens before any other member function invocation on that object and, unless otherwise specified, to ensure that completion of any member function invocation other

§ 17.6.4.10 421

than destruction on such an object happens before destruction of that object. This applies even to objects such as mutexes intended for thread synchronization.  $-end\ note$ 

# 17.6.4.11 Requires paragraph

[res.on.required]

Violation of the preconditions specified in a function's *Requires*: paragraph results in undefined behavior unless the function's *Throws*: paragraph specifies throwing an exception when the precondition is violated.

# 17.6.5 Conforming implementations

[conforming]

#### 17.6.5.1 Overview

[conforming.overview]

- This section describes the constraints upon, and latitude of, implementations of the C++ standard library.
- An implementation's use of headers is discussed in 17.6.5.2, its use of macros in 17.6.5.3, global functions in 17.6.5.4, member functions in 17.6.5.5, data race avoidance in 17.6.5.9, access specifiers in 17.6.5.10, class derivation in 17.6.5.11, and exceptions in 17.6.5.12.

17.6.5.2 Headers [res.on.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.
- <sup>2</sup> 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.5) shall include only their corresponding C++ standard header, as described in 17.6.1.2.

#### 17.6.5.3 Restrictions on macro definitions

[res.on.macro.definitions]

- <sup>1</sup> The names and global function signatures described in 17.6.1.1 are reserved to the implementation.
- <sup>2</sup> All object-like macros defined by the C standard library and described in this Clause as expanding to integral constant expressions are also suitable for use in #if preprocessing directives, unless explicitly stated otherwise.

#### 17.6.5.4 Global and non-member functions

[global.functions]

- <sup>1</sup> 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. 186
- <sup>3</sup> An implementation shall not declare a global or non-member function signature with additional default arguments.
- <sup>4</sup> 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:

```
*out_stream << value;
if (delim != 0)
   *out_stream << delim;
return (*this);
— end note]</pre>
```

#### 17.6.5.5 Member functions

[member.functions]

- It is unspecified whether any member functions in the C++ standard library are defined as inline (7.1.2).
- <sup>2</sup> An implementation may declare additional non-virtual member function signatures within a class:
  - by adding arguments with default values to a member function signature;  $^{187}$  [ Note: An implementation may not add arguments with default values to virtual, global, or non-member functions. end note]

186) A valid C++ program always calls the expected library global or non-member function. An implementation may also define additional global or non-member functions that would otherwise not be called by a valid C++ program.

187) Hence, the address of a member function of a class in the C++ standard library has an unspecified type.

§ 17.6.5.5

— 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.
- A call to a member function signature described in the C++ standard library behaves as if the implementation declares no additional member function signatures. 188

## 17.6.5.6 constexpr functions and constructors

[constexpr.functions]

Within any header that provides any non-defining declarations of constexpr functions or constructors an implementation shall provide corresponding definitions.

# 17.6.5.7 Requirements for stable algorithms

[algorithm.stable]

- When the requirements for an algorithm state that it is "stable" without further elaboration, it means:
  - 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.

#### 17.6.5.8 Reentrancy

[reentrancy]

Except where explicitly specified in this standard, it is implementation-defined which functions in the Standard C++ library may be recursively reentered.

#### 17.6.5.9 Data race avoidance

[res.on.data.races]

- <sup>1</sup> 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.
- <sup>2</sup> 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.
- <sup>3</sup> 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.
- <sup>4</sup> [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.
- Operations on iterators obtained by calling a standard library container or string member function may access the underlying container, but shall not modify it. [Note: In particular, container operations that invalidate iterators conflict with operations on iterators associated with that container. end note]
- Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.
- <sup>8</sup> 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.

§ 17.6.5.9 423

<sup>188)</sup> A valid C++ program always calls the expected library member function, or one with equivalent behavior. An implementation may also define additional member functions that would otherwise not be called by a valid C++ program.

[Note: This allows implementations to parallelize operations if there are no visible side effects. — end note]
17.6.5.10 Protection within classes [protection.within.classes]

<sup>1</sup> It is unspecified whether any function signature or class described in Clauses 18 through 30 and Annex D is a friend of another class in the C++ standard library.

#### 17.6.5.11 Derived classes

[derivation]

- An implementation may derive any class in the C++ standard library from a class with a name reserved to the implementation.
- <sup>2</sup> 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.
- <sup>3</sup> 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. 189

# 17.6.5.12 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. An implementation may strengthen the *exception-specification* for a non-virtual function by adding a non-throwing *noexcept-specification*.
- A function may throw an object of a type not listed in its **Throws** clause if its type is derived from a type named in the **Throws** clause and would be caught by an exception handler for the base type.
- <sup>3</sup> Functions from the C standard library shall not throw exceptions<sup>190</sup> except when such a function calls a program-supplied function that throws an exception.<sup>191</sup>
- <sup>4</sup> Destructor operations defined in the C++ standard library shall not throw exceptions. Every destructor in the C++ standard library shall behave as if it had a non-throwing exception specification. Any other functions defined in the C++ standard library that do not have an exception-specification may throw implementation-defined exceptions unless otherwise specified.<sup>192</sup> An implementation may strengthen this implicit exception-specification by adding an explicit one.<sup>193</sup>

# 17.6.5.13 Restrictions on storage of pointers

# [res.on.pointer.storage]

Objects constructed by the standard library that may hold a user-supplied pointer value or an integer of type std::intptr\_t shall store such values in a traceable pointer location (3.7.4.3). [Note: Other libraries are strongly encouraged to do the same, since not doing so may result in accidental use of pointers that are not safely derived. Libraries that store pointers outside the user's address space should make it appear that they are stored and retrieved from a traceable pointer location. — end note]

#### 17.6.5.14 Value of error codes

[value.error.codes]

Certain functions in the C++ standard library report errors via a std::error\_code (19.5.2.1) object. That object's category() member shall return std::system\_category() for errors originating from the operating system, or a reference to an implementation-defined error\_category object for errors originating

§ 17.6.5.14 424

<sup>189)</sup> There is an implicit exception to this rule for types that are described as synonyms for basic integral types, such as size\_t (18.2) and streamoff (27.5.2).

<sup>190)</sup> That is, the C library functions can all be treated as if they are marked noexcept. This allows implementations to make performance optimizations based on the absence of exceptions at runtime.

<sup>191)</sup> The functions qsort() and bsearch() (25.5) meet this condition.

<sup>192)</sup> In particular, they can report a failure to allocate storage by throwing an exception of type bad\_alloc, or a class derived from bad\_alloc (18.6.2.1). Library implementations should report errors by throwing exceptions of or derived from the standard exception classes (18.6.2.1, 18.8, 19.2).

<sup>193)</sup> That is, an implementation may provide an explicit exception-specification that defines the subset of "any" exceptions thrown by that function. This implies that the implementation may list implementation-defined types in such an exception-specification.

elsewhere. The implementation shall define the possible values of value() for each of these error categories. [Example: For operating systems that are based on POSIX, implementations are encouraged to define the std::system\_category() values as identical to the POSIX errno values, with additional values as defined by the operating system's documentation. Implementations for operating systems that are not based on POSIX are encouraged to define values identical to the operating system's values. For errors that do not originate from the operating system, the implementation may provide enums for the associated values. — end example]

# 17.6.5.15 Moved-from state of library types

[lib.types.movedfrom]

Objects of types defined in the C++ standard library may be moved from (12.8). Move operations may be explicitly specified or implicitly generated. Unless otherwise specified, such moved-from objects shall be placed in a valid but unspecified state.

§ 17.6.5.15 425

# 18 Language support library [language.support]

18.1 General [support.general]

This Clause describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. It also describes the headers that declare these function signatures and define any related types.

The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, support for initializer lists, and other runtime support, as summarized in Table 29.

	Subclause	Header(s)
18.2	Types	<cstddef></cstddef>
		<li><li>imits&gt;</li></li>
18.3	Implementation properties	<climits></climits>
		<cfloat></cfloat>
18.4	Integer types	<cstdint></cstdint>
18.5	Start and termination	<cstdlib></cstdlib>
18.6	Dynamic memory management	<new></new>
18.7	Type identification	<typeinfo></typeinfo>
18.8	Exception handling	<exception></exception>
18.9	Initializer lists	<pre><initializer_list></initializer_list></pre>
		<csignal></csignal>
		<csetjmp></csetjmp>
		<cstdalign></cstdalign>
18.10	Other runtime support	<cstdarg></cstdarg>
		<cstdbool></cstdbool>
		<cstdlib></cstdlib>
		<ctime></ctime>

Table 29 — Language support library summary

18.2 Types [support.types]

Table 30 describes the header <cstddef>.

Table 30 — Header <cstddef> synopsis

Type	Name(s)	
Macros:	NULL	offsetof
Types:	ptrdiff_t	size_t
	max_align_t	${\tt nullptr\_t}$

§ 18.2 426

- The contents are the same as the Standard C library header <stddef.h>, with the following changes:
- <sup>3</sup> The macro NULL is an implementation-defined C++ null pointer constant in this International Standard (4.10).<sup>194</sup>
- 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. The expression offsetof(type, member-designator) is never type-dependent (14.6.2.2) and it is value-dependent (14.6.2.3) if and only if type is dependent. The result of applying the offsetof macro to a field that is a static data member or a function member is undefined. No operation invoked by the offsetof macro shall throw an exception and noexcept(offsetof(type, member-designator)) shall be true.
- <sup>5</sup> The type ptrdiff\_t is an implementation-defined signed integer type that can hold the difference of two subscripts in an array object, as described in 5.7.
- <sup>6</sup> The type size\_t is an implementation-defined unsigned integer type that is large enough to contain the size in bytes of any object.
- 7 [Note: It is recommended that implementations choose types for ptrdiff\_t and size\_t whose integer conversion ranks (4.13) are no greater than that of signed long int unless a larger size is necessary to contain all the possible values. end note]
- <sup>8</sup> 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.
- 9 nullptr\_t is defined as follows:

```
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 Ivalue can be taken. — end note]

SEE ALSO: Alignment (3.11), Sizeof (5.3.3), Additive operators (5.7), Free store (12.5), and ISO C 7.1.6.

#### 18.3 Implementation properties

[support.limits]

# 18.3.1 In general

[support.limits.general]

The headers imits> (18.3.2), <climits>, and <cfloat> (18.3.3) supply characteristics of implementation-dependent arithmetic types (3.9.1).

# 18.3.2 Numeric limits

[limits]

# 18.3.2.1 Class template numeric\_limits

[limits.numeric]

- The numeric\_limits class template provides a C++ program with information about various properties of the implementation's representation of the arithmetic types.
- <sup>2</sup> 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.
- <sup>3</sup> 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.
- 4 Non-arithmetic standard types, such as complex<T> (26.4.2), shall not have specializations.

#### 18.3.2.2 Header imits> synopsis

[limits.syn]

```
namespace std {
  template<class T> class numeric_limits;
  enum float_round_style;
  enum float_denorm_style;
```

§ 18.3.2.2

<sup>194)</sup> Possible definitions include 0 and 0L, but not (void\*)0.

<sup>195)</sup> 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<bool>;
    template<> class numeric_limits<char>;
   template<> class numeric_limits<signed char>;
    template<> class numeric_limits<unsigned char>;
   template<> class numeric_limits<char16_t>;
    template<> class numeric_limits<char32_t>;
    template<> class numeric_limits<wchar_t>;
   template<> class numeric_limits<short>;
    template<> class numeric_limits<int>;
    template<> class numeric_limits<long>;
    template<> class numeric_limits<long long>;
    template<> class numeric_limits<unsigned short>;
    template<> class numeric_limits<unsigned int>;
    template<> class numeric_limits<unsigned long>;
    template<> class numeric_limits<unsigned long long>;
    template<> class numeric_limits<float>;
    template<> class numeric_limits<double>;
    template<> class numeric_limits<long double>;
18.3.2.3 Class template numeric limits
                                                                                   [numeric.limits]
 namespace std {
   template<class T> class numeric_limits {
   public:
     static constexpr bool is_specialized = false;
     static constexpr T min() noexcept { return T(); }
     static constexpr T max() noexcept { return T(); }
     static constexpr T lowest() noexcept { return T(); }
     static constexpr int digits = 0;
     static constexpr int digits10 = 0;
     static constexpr int max_digits10 = 0;
     static constexpr bool is_signed = false;
      static constexpr bool is_integer = false;
      static constexpr bool is_exact = false;
      static constexpr int radix = 0;
      static constexpr T epsilon() noexcept { return T(); }
     static constexpr T round_error() noexcept { return T(); }
     static constexpr int min_exponent = 0;
      static constexpr int min_exponent10 = 0;
      static constexpr int max_exponent = 0;
      static constexpr int max_exponent10 = 0;
     static constexpr bool has_infinity = false;
     static constexpr bool has_quiet_NaN = false;
      static constexpr bool has_signaling_NaN = false;
      static constexpr float_denorm_style has_denorm = denorm_absent;
      static constexpr bool has_denorm_loss = false;
      static constexpr T infinity() noexcept { return T(); }
      static constexpr T quiet_NaN() noexcept { return T(); }
     static constexpr T signaling_NaN() noexcept { return T(); }
```

§ 18.3.2.3 428

```
static constexpr T denorm_min() noexcept { return T(); }
          static constexpr bool is_iec559 = false;
          static constexpr bool is_bounded = false;
          static constexpr bool is_modulo = false;
          static constexpr bool traps = false;
          static constexpr bool tinyness before = false;
          static constexpr float_round_style round_style = round_toward_zero;
        };
        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>;
   The default numeric limits<T> template shall have all members, but with 0 or false values.
   The value of each member of a specialization of numeric_limits on a cv-qualified type cv T shall be equal
    to the value of the corresponding member of the specialization on the unqualified type T.
    18.3.2.4 numeric_limits members
                                                                                  [numeric.limits.members]
    static constexpr T min() noexcept;
         Minimum finite value. 196
1
2
         For floating types with denormalization, returns the minimum positive normalized value.
3
         Meaningful for all specializations in which is bounded != false, or is bounded == false && is -
         signed == false.
    static constexpr T max() noexcept;
         Maximum finite value. 197
4
5
         Meaningful for all specializations in which is_bounded != false.
    static constexpr T lowest() noexcept;
6
         A finite value x such that there is no other finite value y where y < x.^{198}
7
         Meaningful for all specializations in which is_bounded != false.
    static constexpr int digits;
8
         Number of radix digits that can be represented without change.
9
         For integer types, the number of non-sign bits in the representation.
10
         For floating point types, the number of radix digits in the mantissa. 199
    static constexpr int digits10;
    196) Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc.
    197) Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.
    198) lowest() is necessary because not all floating-point representations have a smallest (most negative) value that is the
    negative of the largest (most positive) finite value.
    199) Equivalent to FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG.
    § 18.3.2.4
                                                                                                           429
```

```
11
          Number of base 10 digits that can be represented without change.<sup>200</sup>
12
          Meaningful for all specializations in which is_bounded != false.
    static constexpr int max_digits10;
13
          Number of base 10 digits required to ensure that values which differ are always differentiated.
14
          Meaningful for all floating point types.
    static constexpr bool is_signed;
15
          True if the type is signed.
16
          Meaningful for all specializations.
    static constexpr bool is_integer;
17
          True if the type is integer.
18
          Meaningful for all specializations.
    static constexpr bool is_exact;
19
          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.
20
          Meaningful for all specializations.
    static constexpr int radix;
21
          For floating types, specifies the base or radix of the exponent representation (often 2).<sup>201</sup>
          For integer types, specifies the base of the representation.<sup>202</sup>
22
23
          Meaningful for all specializations.
    static constexpr T epsilon() noexcept;
24
          Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.<sup>203</sup>
25
          Meaningful for all floating point types.
    static constexpr T round_error() noexcept;
          Measure of the maximum rounding error.<sup>204</sup>
26
    static constexpr int min_exponent;
    200) Equivalent to FLT_DIG, DBL_DIG, LDBL_DIG.
    201) Equivalent to FLT_RADIX.
    202) Distinguishes types with bases other than 2 (e.g. BCD).
    203) Equivalent to FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON.
    204) Rounding error is described in ISO/IEC 10967-1 Language independent arithmetic - Part 1 Section 5.2.8 and Annex A
    Rationale Section A.5.2.8 - Rounding constants.
```

430

§ 18.3.2.4

27 Minimum negative integer such that radix raised to the power of one less than that integer is a normalized floating point number.<sup>205</sup> 28 Meaningful for all floating point types. static constexpr int min\_exponent10; 29 Minimum negative integer such that 10 raised to that power is in the range of normalized floating point numbers.<sup>206</sup> 30 Meaningful for all floating point types. static constexpr int max\_exponent; 31 Maximum positive integer such that radix raised to the power one less than that integer is a representable finite floating point number.<sup>207</sup> 32 Meaningful for all floating point types. static constexpr int max\_exponent10; 33 Maximum positive integer such that 10 raised to that power is in the range of representable finite floating point numbers.<sup>208</sup> 34 Meaningful for all floating point types. static constexpr bool has\_infinity; 35 True if the type has a representation for positive infinity. 36 Meaningful for all floating point types. 37 Shall be true for all specializations in which is iec559 != false. static constexpr bool has\_quiet\_NaN; True if the type has a representation for a quiet (non-signaling) "Not a Number." <sup>209</sup> 38 Meaningful for all floating point types. 39 40 Shall be true for all specializations in which is iec559 != false. static constexpr bool has\_signaling\_NaN; True if the type has a representation for a signaling "Not a Number." <sup>210</sup> 41 42 Meaningful for all floating point types. 43 Shall be true for all specializations in which is\_iec559 != false. 205) Equivalent to FLT\_MIN\_EXP, DBL\_MIN\_EXP, LDBL\_MIN\_EXP. 206) Equivalent to FLT\_MIN\_10\_EXP, DBL\_MIN\_10\_EXP, LDBL\_MIN\_10\_EXP. 207) Equivalent to FLT\_MAX\_EXP, DBL\_MAX\_EXP, LDBL\_MAX\_EXP.

§ 18.3.2.4 431

208) Equivalent to FLT\_MAX\_10\_EXP, DBL\_MAX\_10\_EXP, LDBL\_MAX\_10\_EXP.

209) Required by LIA-1. 210) Required by LIA-1.

static constexpr float\_denorm\_style has\_denorm;

```
denorm_present if the type allows denormalized values (variable number of exponent bits) 211, denorm_-
44
          absent if the type does not allow denormalized values, and denorm indeterminate if it is indetermi-
          nate at compile time whether the type allows denormalized values.
45
          Meaningful for all floating point types.
    static constexpr bool has_denorm_loss;
          True if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.<sup>212</sup>
46
    static constexpr T infinity() noexcept;
          Representation of positive infinity, if available. ^{213}
47
48
          Meaningful for all specializations for which has_infinity != false. Required in specializations for
          which is_iec559 != false.
    static constexpr T quiet_NaN() noexcept;
          Representation of a quiet "Not a Number," if available. 214
49
50
          Meaningful for all specializations for which has_quiet_NaN != false. Required in specializations for
          which is_iec559 != false.
    static constexpr T signaling_NaN() noexcept;
51
          Representation of a signaling "Not a Number," if available. <sup>215</sup>
52
          Meaningful for all specializations for which has_signaling_NaN != false. Required in specializations
          for which is_iec559 != false.
    static constexpr T denorm_min() noexcept;
          Minimum positive denormalized value.<sup>216</sup>
53
          Meaningful for all floating point types.
54
55
          In specializations for which has_denorm == false, returns the minimum positive normalized value.
    static constexpr bool is_iec559;
56
          True if and only if the type adheres to IEC 559 standard.<sup>217</sup>
57
          Meaningful for all floating point types.
    211) Required by LIA-1.
    212) See IEC 559.
    213) Required by LIA-1.
    214) Required by LIA-1.
    215) Required by LIA-1.
    216) Required by LIA-1.
    217) International Electrotechnical Commission standard 559 is the same as IEEE 754.
```

§ 18.3.2.4 432

```
static constexpr bool is_bounded;
          True if the set of values representable by the type is finite. <sup>218</sup> [Note: All fundamental types (3.9.1)
58
          are bounded. This member would be false for arbitrary precision types. — end\ note
          Meaningful for all specializations.
59
    static constexpr bool is_modulo;
60
          True if the type is modulo.<sup>219</sup> A type is modulo if, for any operation involving +, -, or * on values
          of that type whose result would fall outside the range [min(), max()], the value returned differs from
          the true value by an integer multiple of max() - min() + 1.
61
          On most machines, this is false for floating types, true for unsigned integers, and true for signed
          integers.
62
          Meaningful for all specializations.
    static constexpr bool traps;
63
          true if, at program startup, there exists a value of the type that would cause an arithmetic operation
          using that value to trap.^{220}
64
          Meaningful for all specializations.
    static constexpr bool tinyness_before;
65
          true if tinyness is detected before rounding.<sup>221</sup>
66
          Meaningful for all floating point types.
    static constexpr float_round_style round_style;
67
          The rounding style for the type.<sup>222</sup>
          Meaningful for all floating point types. Specializations for integer types shall return round_toward_-
68
               Type float_round_style
                                                                                                    [round.style]
    18.3.2.5
      namespace std {
        enum float_round_style {
          round_indeterminate
                                        = -1,
          round_toward_zero
                                          Ο,
          round_to_nearest
                                           1,
          round_toward_infinity
                                          2,
          round_toward_neg_infinity = 3
        };
      }
    218) Required by LIA-1.
    219) Required by LIA-1.
    220) Required by LIA-1.
    221) Refer to IEC 559. Required by LIA-1.
```

§ 18.3.2.5 433

222) Equivalent to FLT\_ROUNDS. Required by LIA-1.

- The rounding mode for floating point arithmetic is characterized by the values:
  - round\_indeterminate if the rounding style is indeterminable
  - round\_toward\_zero if the rounding style is toward zero
  - round\_to\_nearest if the rounding style is to the nearest representable value
  - round\_toward\_infinity if the rounding style is toward infinity
  - round\_toward\_neg\_infinity if the rounding style is toward negative infinity

## 18.3.2.6 Type float\_denorm\_style

[denorm.style]

```
namespace std {
  enum float_denorm_style {
    denorm_indeterminate = -1,
    denorm_absent = 0,
    denorm_present = 1
  };
}
```

- <sup>1</sup> The presence or absence of denormalization (variable number of exponent bits) is characterized by the values:
  - denorm\_indeterminate if it cannot be determined whether or not the type allows denormalized values
    - denorm\_absent if the type does not allow denormalized values
    - denorm\_present if the type does allow denormalized values

# 18.3.2.7 numeric\_limits specializations

[numeric.special]

- 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.
- <sup>2</sup> [Example:

```
namespace std {
  template<> class numeric_limits<float> {
  public:
    static constexpr bool is_specialized = true;
    inline static constexpr float min() noexcept { return 1.17549435E-38F; }
    inline static constexpr float max() noexcept { return 3.40282347E+38F; }
    inline static constexpr float lowest() noexcept { return -3.40282347E+38F; }
    static constexpr int digits = 24;
    static constexpr int digits10 = 6;
    static constexpr int max_digits10 = 9;
   static constexpr bool is_signed = true;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact = false;
    static constexpr int radix = 2;
    inline static constexpr float epsilon() noexcept
                                                         { return 1.19209290E-07F; }
    inline static constexpr float round_error() noexcept { return 0.5F; }
    static constexpr int min_exponent
                                      = -125;
    static constexpr int min_exponent10 = - 37;
```

§ 18.3.2.7

```
static constexpr int max_exponent = +128;
         static constexpr int max_exponent10 = + 38;
         static constexpr bool has_infinity
                                                       = true:
         static constexpr bool has_quiet_NaN
                                                       = true;
         static constexpr bool has_signaling_NaN
                                                      = true;
         static constexpr float_denorm_style has_denorm = denorm_absent;
         static constexpr bool has_denorm_loss
                                                        = false;
         inline static constexpr float infinity()
                                                       noexcept { return value; }
         inline static constexpr float quiet_NaN()
                                                       noexcept { return value; }
         inline static constexpr float signaling_NaN() noexcept { return value; }
         inline static constexpr float denorm_min()
                                                       noexcept { return min(); }
         static constexpr bool is_iec559 = true;
         static constexpr bool is_bounded = true;
         static constexpr bool is_modulo = false;
         static constexpr bool traps
                                         = true;
         static constexpr bool tinyness_before = true;
         static constexpr float_round_style round_style = round_to_nearest;
       };
     }
    - end example ]
<sup>3</sup> The specialization for bool shall be provided as follows:
     namespace std {
        template<> class numeric_limits<bool> {
        public:
          static constexpr bool is_specialized = true;
          static constexpr bool min() noexcept { return false; }
          static constexpr bool max() noexcept { return true; }
          static constexpr bool lowest() noexcept { return false; }
          static constexpr int digits = 1;
          static constexpr int digits10 = 0;
          static constexpr int max_digits10 = 0;
          static constexpr bool is_signed = false;
          static constexpr bool is_integer = true;
          static constexpr bool is_exact = true;
          static constexpr int radix = 2;
          static constexpr bool epsilon() noexcept { return 0; }
          static constexpr bool round_error() noexcept { return 0; }
          static constexpr int min_exponent = 0;
          static constexpr int min_exponent10 = 0;
          static constexpr int max_exponent = 0;
          static constexpr int max_exponent10 = 0;
          static constexpr bool has_infinity = false;
          static constexpr bool has_quiet_NaN = false;
          static constexpr bool has_signaling_NaN = false;
          static constexpr float_denorm_style has_denorm = denorm_absent;
          static constexpr bool has_denorm_loss = false;
```

§ 18.3.2.7

```
static constexpr bool infinity() noexcept { return 0; }
static constexpr bool quiet_NaN() noexcept { return 0; }
static constexpr bool signaling_NaN() noexcept { return 0; }
static constexpr bool denorm_min() noexcept { return 0; }

static constexpr bool is_iec559 = false;
static constexpr bool is_bounded = true;
static constexpr bool is_modulo = false;

static constexpr bool traps = false;
static constexpr bool tinyness_before = false;
static constexpr float_round_style round_style = round_toward_zero;
};
}
```

# 18.3.3 C library

[c.limits]

Table 31 describes the header <climits>.

Table 31 — Header <climits> synopsis

Type			Name(s)		
Values:					
CHAR_BIT	INT_MAX	LONG_MAX	SCHAR_MIN	SHRT_MIN	ULLONG_MAX
CHAR_MAX	LLONG_MAX	LONG_MIN	SCHAR_MAX	UCHAR_MAX	ULONG_MAX
CHAR_MIN	LLONG_MIN	MB_LEN_MAX	SHRT_MAX	UINT_MAX	USHRT_MAX
INT_MIN					

- 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]
- <sup>3</sup> Table 32 describes the header <cfloat>.

Table 32 — Header <cfloat> synopsis

Type		Name(s)	
Values:			
DBL_DIG	DBL_MIN_EXP	FLT_MAX_EXP	LDBL_MANT_DIG
DBL_EPSILON	DECIMAL_DIG	FLT_MIN	LDBL_MAX_10_EXP
DBL_MANT_DIG	FLT_DIG	FLT_MIN_10_EXP	LDBL_MAX_EXP
DBL_MAX	FLT_EPSILON	FLT_MIN_EXP	LDBL_MAX
DBL_MAX_10_EXP	FLT_EVAL_METHOD	FLT_RADIX	LDBL_MIN
DBL_MAX_EXP	FLT_MANT_DIG	FLT_ROUNDS	LDBL_MIN_10_EXP
DBL_MIN	FLT_MAX	LDBL_DIG	LDBL_MIN_EXP
DBL_MIN_10_EXP	FLT_MAX_10_EXP	LDBL_EPSILON	

<sup>4</sup> The contents are the same as the Standard C library header <float.h>. SEE ALSO: ISO C 7.1.5, 5.2.4.2.2, 5.2.4.2.1.

# 18.4 Integer types

[cstdint]

18.4.1 Header <cstdint> synopsis

[cstdint.syn]

§ 18.4.1

```
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
                                                      // optional
       typedef unsigned integer type uint8_t;
                                                      // optional
       typedef unsigned integer type uint16_t;
                                                      // optional
       typedef unsigned integer type uint32_t;
       typedef unsigned integer type uint64_t;
                                                      // optional
       typedef unsigned integer type uint_fast8_t;
       typedef unsigned integer type uint_fast16_t;
       typedef unsigned integer type uint_fast32_t;
       typedef unsigned integer type uint_fast64_t;
       typedef unsigned integer type uint_least8_t;
       typedef unsigned integer type uint_least16_t;
       typedef unsigned integer type uint_least32_t;
       typedef unsigned integer type uint_least64_t;
       typedef unsigned integer type uintmax_t;
                                                      // optional
       typedef unsigned integer type uintptr_t;
     } // namespace std
<sup>1</sup> The header also defines numerous macros of the form:
       INT_[FAST LEAST] {8 16 32 64}_MIN
       [U]INT_[FAST LEAST]{8 16 32 64}_MAX
       INT{MAX PTR}_MIN
       [U] INT{MAX PTR}_MAX
       {PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
       SIZE_MAX
      plus function macros of the form:
       [U]INT{8 16 32 64 MAX}_C
```

The header defines all functions, types, and macros the same as 7.18 in the C standard. [Note: The macros defined by <cstdint> are provided unconditionally. In particular, the symbols \_\_STDC\_LIMIT\_MACROS and \_\_STDC\_CONSTANT\_MACROS (mentioned in footnotes 219, 220, and 222 in the C standard) play no role in

§ 18.5

Type		Name(s)	
Macros:	EXIT_FAILURE	EXIT_SUCCESS	
Functions:	_Exit	abort	atexit
	at_quick_exit	exit	quick_exit

C++. — end note]

3

#### 18.5 Start and termination

[support.start.term]

- Table 33 describes some of the contents of the header <cstdlib>.
- <sup>2</sup> The contents are the same as the Standard C library header <stdlib.h>, with the following changes:

[[noreturn]] void \_Exit(int status) noexcept;

- The function Exit(int status) has additional behavior in this International Standard:
  - The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to atexit() (3.6.3).

[[noreturn]] void abort(void) noexcept;

- 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 functions passed to atexit() (3.6.3).

```
extern "C" int atexit(void (*f)(void)) noexcept;
extern "C++" int atexit(void (*f)(void)) noexcept;
```

- Effects: The atexit() functions register the function pointed to by f to be called without arguments at normal program termination. It is unspecified whether a call to atexit() that does not happen before (1.10) a call to exit() will succeed. [Note: The atexit() functions do not introduce a data race (17.6.5.9). end note]
- 6 Implementation limits: The implementation shall support the registration of at least 32 functions.
- 7 Returns: The atexit() function returns zero if the registration succeeds, non-zero if it fails.

[[noreturn]] void exit(int status)

- The function exit() has additional behavior in this International Standard:
  - 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.<sup>223</sup> See 3.6.3 for the order of destructions and calls. (Automatic objects are not destroyed as a result of calling exit().)<sup>224</sup>

If control leaves a registered function called by exit because the function does not provide a handler for a thrown exception, std::terminate() shall be called (15.5.1).

§ 18.5 438

<sup>223)</sup> A function is called for every time it is registered.

<sup>224)</sup> 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().

— 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. 225

```
extern "C" int at_quick_exit(void (*f)(void)) noexcept;
extern "C++" int at_quick_exit(void (*f)(void)) noexcept;
```

Effects: The at\_quick\_exit() functions register the function pointed to by f to be called without arguments when quick\_exit is called. It is unspecified whether a call to at\_quick\_exit() that does not happen before (1.10) all calls to quick\_exit will succeed. [Note: The at\_quick\_exit() functions do not introduce a data race (17.6.5.9). — end note] [Note: The order of registration may be indeterminate if at\_quick\_exit was called from more than one thread. — end note] [Note: The at\_quick\_exit registrations are distinct from the atexit registrations, and applications may need to call both registration functions with the same argument. — end note]

- Implementation limits: The implementation shall support the registration of at least 32 functions.
- 11 Returns: Zero if the registration succeeds, non-zero if it fails.

[[noreturn]] void quick\_exit(int status) noexcept;

Effects: Functions registered by calls to at\_quick\_exit are called in the reverse order of their registration, except that a function shall be called after any previously registered functions that had already been called at the time it was registered. Objects shall not be destroyed as a result of calling quick\_exit. If control leaves a registered function called by quick\_exit because the function does not provide a handler for a thrown exception, std::terminate() shall be called. [Note: at\_quick\_exit may call a registered function from a different thread than the one that registered it, so registered functions should not rely on the identity of objects with thread storage duration. —end note] After calling registered functions, quick\_exit shall call \_Exit(status). [Note: The standard file buffers are not flushed. See: ISO C 7.20.4.4. —end note]

SEE ALSO: 3.6, 3.6.3, ISO C 7.10.4.

# 18.6 Dynamic memory management

[support.dynamic]

The header <new> defines several functions that manage the allocation of dynamic storage in a program. It also defines components for reporting storage management errors.

# Header <new> synopsis

9

10

```
namespace std {
  class bad_alloc;
  class bad_array_new_length;
  struct nothrow_t {};
  extern const nothrow_t nothrow;
  typedef void (*new_handler)();
  new_handler get_new_handler() noexcept;
  new_handler set_new_handler(new_handler new_p) noexcept;
}
```

225) The macros  ${\tt EXIT\_FAILURE}$  and  ${\tt EXIT\_SUCCESS}$  are defined in  ${\tt <cstdlib>}.$ 

§ 18.6

```
void* operator new(std::size_t size);
void* operator new(std::size_t size, const std::nothrow_t&) noexcept;
void operator delete(void* ptr) noexcept;
void operator new[](std::size_t size);
void* operator new[](std::size_t size);
void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
void operator delete[](void* ptr) noexcept;
void operator delete[](void* ptr) noexcept;
void operator new (std::size_t size, void* ptr) noexcept;
void* operator new[](std::size_t size, void* ptr) noexcept;
void operator delete (void* ptr, void*) noexcept;
void operator delete (void* ptr, void*) noexcept;
void operator delete[](void* ptr, void*) noexcept;
```

# 18.6.1 Storage allocation and deallocation

[new.delete]

Except where otherwise specified, the provisions of (3.7.4) apply to the library versions of operator new and operator delete.

#### 18.6.1.1 Single-object forms

[new.delete.single]

```
void* operator new(std::size_t size);
```

- 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.
- 4 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 current new\_handler (18.6.2.5) is a null pointer value, throws bad\_alloc.
  - Otherwise, the function calls the current new\_handler function (18.6.2.3). If the called function returns, the loop repeats.
  - The loop terminates when an attempt to allocate the requested storage is successful or when a called new\_handler function does not return.

void\* operator new(std::size\_t size, const std::nothrow\_t&) noexcept;

- Effects: Same as above, except that it is called by a placement version of a new-expression when a C++ program prefers a null pointer result as an error indication, instead of a bad\_alloc exception.
- Replaceable: a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.
- Required behavior: Return a non-null pointer to suitably aligned storage (3.7.4), or else return a null pointer. This nothrow version of operator new returns a pointer obtained as if acquired from the (possibly replaced) ordinary version. This requirement is binding on a replacement version of this function.

§ 18.6.1.1 440

Be a Default behavior: Calls operator new(size). If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

9 [Example:

```
T* p1 = new T;  // throws bad_alloc if it fails
T* p2 = new(nothrow) T;  // returns 0 if it fails
-- end example
```

void operator delete(void\* ptr) noexcept;

- Effects: The deallocation function (3.7.4.2) called by a delete-expression to render the value of ptr invalid.
- Replaceable: a C++ program may define a function with 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>.

void operator delete(void\* ptr, const std::nothrow\_t&) noexcept;

- Effects: The deallocation function (3.7.4.2) called by the implementation to render the value of ptr invalid when the constructor invoked from a nothrow placement version of the new-expression throws an exception.
- Replaceable: a C++ program may define a function with 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.
- 19 Default behavior: calls operator delete(ptr).

#### **18.6.1.2** Array forms

1

[new.delete.array]

```
void* operator new[](std::size_t size);
```

- Effects: The allocation function (3.7.4.1) called by the array form of a new-expression (5.3.4) to allocate size bytes of storage suitably aligned to represent any array object of that size or smaller.<sup>226</sup>
- 2 Replaceable: a C++ program can define a function with this function signature that displaces the default version defined by the C++ standard library.

§ 18.6.1.2 441

<sup>226)</sup> 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.

Required behavior: Same as for operator new(std::size\_t). This requirement is binding on a replacement version of this function.

4 Default behavior: Returns operator new(size).

void\* operator new[](std::size\_t size, const std::nothrow\_t&) noexcept;

- Effects: Same as above, except that it is called by a placement version of a new-expression when a C++ program prefers a null pointer result as an error indication, instead of a bad\_alloc exception.
- Replaceable: a C++ program can define a function with this function signature that displaces the default version defined by the C++ standard library.
- <sup>7</sup> 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.
- 8 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) noexcept;

- 9 Effects: The deallocation function (3.7.4.2) called by the array form of a delete-expression to render the value of ptr invalid.
- Replaceable: a C++ program can define a function with 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.
- 13 Default behavior: Calls operator delete(ptr).

void operator delete[](void\* ptr, const std::nothrow\_t&) noexcept;

- Effects: The deallocation function (3.7.4.2) called by the implementation to render the value of ptr invalid when the constructor invoked from a nothrow placement version of the array new-expression throws an exception.
- Replaceable: a C++ program may define a function with 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.
- 17 Default behavior: calls operator delete[](ptr).

#### 18.6.1.3 Placement forms

[new.delete.placement]

These functions are reserved, a C++ program may not define functions that displace the versions in the Standard C++ library (17.6.4). The provisions of (3.7.4) do not apply to these reserved placement forms of operator new and operator delete.

void\* operator new(std::size\_t size, void\* ptr) noexcept;

§ 18.6.1.3 442

```
2
          Returns: ptr.
3
          Remarks: Intentionally performs no other action.
4
         Example: This can be useful for constructing an object at a known address:
           void* place = operator new(sizeof(Something));
           Something* p = new (place) Something();
          - end example]
    void* operator new[](std::size_t size, void* ptr) noexcept;
5
          Returns: ptr.
6
          Remarks: Intentionally performs no other action.
    void operator delete(void* ptr, void*) noexcept;
7
         Effects: Intentionally performs no action.
8
         Requires: If an implementation has strict pointer safety (3.7.4.3) then ptr shall be a safely-derived
         pointer.
9
         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*) noexcept;
10
          Effects: Intentionally performs no action.
11
          Requires: If an implementation has strict pointer safety (3.7.4.3) then ptr shall be a safely-derived
         pointer.
12
         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
                                                                                     [new.delete.dataraces]
   For purposes of determining the existence of data races, the library versions of operator new, user replace-
    ment versions of global operator new, and the C standard library functions calloc and malloc shall behave
    as though they accessed and modified only the storage referenced by the return value. The library versions
    of operator delete, user replacement versions of operator delete, and the C standard library function
    free shall behave as though they accessed and modified only the storage referenced by their first argument.
    The C standard library function realloc shall behave as though it accessed and modified only the storage
    referenced by its first argument and by its return value. 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.
              Storage allocation errors
                                                                                              [alloc.errors]
    18.6.2
                                                                                                  [bad.alloc]
    18.6.2.1
               Class bad_alloc
      namespace std {
        class bad_alloc : public exception {
        public:
```

§ 18.6.2.1 443

bad\_alloc() noexcept;

bad\_alloc(const bad\_alloc&) noexcept;

bad\_alloc& operator=(const bad\_alloc&) noexcept;

```
virtual const char* what() const noexcept;
       };
   The class bad_alloc defines the type of objects thrown as exceptions by the implementation to report a
   failure to allocate storage.
   bad_alloc() noexcept;
2
        Effects: Constructs an object of class bad_alloc.
3
        Remarks: The result of calling what() on the newly constructed object is implementation-defined.
       bad_alloc(const bad_alloc&) noexcept;
       bad_alloc& operator=(const bad_alloc&) noexcept;
4
        Effects: Copies an object of class bad_alloc.
   virtual const char* what() const noexcept;
5
        Returns: An implementation-defined NTBS.
                                                                                         [new.badlength]
   18.6.2.2
              Class bad_array_new_length
     namespace std {
       class bad_array_new_length : public bad_alloc {
         bad_array_new_length() noexcept;
     }
  The class bad_array_new_length defines the type of objects thrown as exceptions by the implementation
   to report an attempt to allocate an array of size less than zero or greater than an implementation-defined
   limit (5.3.4).
   bad_array_new_length() noexcept;
2
        Effects: constructs an object of class bad_array_new_length.
3
        Remarks: the result of calling what () on the newly constructed object is implementation-defined.
   18.6.2.3
              Type new_handler
                                                                                            [new.handler]
   typedef void (*new_handler)();
1
        The type of a handler function to be called by operator new() or operator new[]() (18.6.1) when
        they cannot satisfy a request for additional storage.
2
        Required behavior: A new_handler shall perform one of the following:
          — 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;
```

§ 18.6.2.3

```
18.6.2.4 set_new_handler
                                                                                       [set.new.handler]
   new_handler set_new_handler(new_handler new_p) noexcept;
1
        Effects: Establishes the function designated by new_p as the current new_handler.
2
        Returns: The previous new_handler.
3
        Remarks: The initial new_handler is a null pointer.
                                                                                       [get.new.handler]
   18.6.2.5
             get_new_handler
   new_handler get_new_handler() noexcept;
1
        Returns: The current new_handler. [Note: This may be a null pointer value. —end note]
         Type identification
                                                                                          [support.rtti]
  The header <typeinfo> defines a type associated with type information generated by the implementation.
   It also defines two types for reporting dynamic type identification errors.
      Header <typeinfo> synopsis
     namespace std {
       class type_info;
       class bad_cast;
       class bad_typeid;
      SEE ALSO: 5.2.7, 5.2.8.
                                                                                             [type.info]
   18.7.1 Class type_info
     namespace std {
       class type_info {
       public:
         virtual ~type_info();
         bool operator==(const type_info& rhs) const noexcept;
         bool operator!=(const type_info& rhs) const noexcept;
         bool before(const type_info& rhs) const noexcept;
         size_t hash_code() const noexcept;
         const char* name() const noexcept;
         type_info(const type_info& rhs) = delete;
                                                                // cannot be copied
         type_info% operator=(const type_info% rhs) = delete; // cannot be copied
       };
  The class type_info describes type information generated by the implementation. Objects of this class
   effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for
   equality or collating order. The names, encoding rule, and collating sequence for types are all unspecified
   and may differ between programs.
   bool operator==(const type_info& rhs) const noexcept;
2
        Effects: Compares the current object with rhs.
3
        Returns: true if the two values describe the same type.
   bool operator!=(const type_info& rhs) const noexcept;
4
        Returns: !(*this == rhs).
   § 18.7.1
                                                                                                      445
```

```
bool before(const type_info& rhs) const noexcept;
5
         Effects: Compares the current object with rhs.
6
         Returns: true if *this precedes rhs in the implementation's collation order.
    size_t hash_code() const noexcept;
7
         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.
8
         Remark: an implementation should return different values for two type_info objects which do not
         compare equal.
    const char* name() const noexcept;
9
         Returns: An implementation-defined NTBS.
10
         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)
                                                                                                 [bad.cast]
    18.7.2
             Class bad_cast
      namespace std {
        class bad_cast : public exception {
        public:
          bad_cast() noexcept;
          bad_cast(const bad_cast&) noexcept;
          bad_cast& operator=(const bad_cast&) noexcept;
          virtual const char* what() const noexcept;
        };
      }
   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() noexcept;
2
         Effects: Constructs an object of class bad_cast.
3
         Remarks: The result of calling what () on the newly constructed object is implementation-defined.
        bad_cast(const bad_cast&) noexcept;
        bad_cast& operator=(const bad_cast&) noexcept;
4
         Effects: Copies an object of class bad_cast.
    virtual const char* what() const noexcept;
5
         Returns: An implementation-defined NTBS.
6
         Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion
         and display as a wstring (21.3, 22.4.1.4)
```

§ 18.7.2 446

# 18.7.3 Class bad\_typeid

[bad.typeid]

```
namespace std {
  class bad_typeid : public exception {
  public:
    bad_typeid() noexcept;
    bad_typeid(const bad_typeid&) noexcept;
    bad_typeid& operator=(const bad_typeid&) noexcept;
    virtual const char* what() const noexcept;
  };
}
```

<sup>1</sup> The class bad\_typeid defines the type of objects thrown as exceptions by the implementation to report a null pointer in a *typeid* expression (5.2.8).

```
bad_typeid() noexcept;
```

- 2 Effects: Constructs an object of class bad\_typeid.
- Remarks: The result of calling what() on the newly constructed object is implementation-defined.

```
bad_typeid(const bad_typeid&) noexcept;
bad_typeid& operator=(const bad_typeid&) noexcept;
```

4 Effects: Copies an object of class bad\_typeid.

```
virtual const char* what() const noexcept;
```

- 5 Returns: An implementation-defined NTBS.
- 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

[support.exception]

The header **<exception>** defines several types and functions related to the handling of exceptions in a C++ program.

#### Header <exception> synopsis

```
namespace std {
  class exception;
  class bad_exception;
  class nested_exception;

  typedef void (*unexpected_handler)();
  unexpected_handler get_unexpected() noexcept;
  unexpected_handler set_unexpected(unexpected_handler f) noexcept;
  [[noreturn]] void unexpected();

  typedef void (*terminate_handler)();
  terminate_handler get_terminate() noexcept;
  terminate_handler set_terminate(terminate_handler f) noexcept;
  [[noreturn]] void terminate() noexcept;

  bool uncaught_exception() noexcept;

  typedef unspecified exception_ptr;
```

§ 18.8 447

exception\_ptr current\_exception() noexcept;

3

4

5

6

7

9

§ 18.8.1

```
[[noreturn]] void rethrow_exception(exception_ptr p);
       template<class E> exception_ptr make_exception_ptr(E e) noexcept;
       [[noreturn]] template <class T> void throw_with_nested(T&& t);
       template <class E> void rethrow_if_nested(const E& e);
      See also: 15.5.
                                                                                              [exception]
   18.8.1
             Class exception
     namespace std {
       class exception {
       public:
         exception() noexcept;
         exception(const exception&) noexcept;
         exception& operator=(const exception&) noexcept;
         virtual ~exception();
         virtual const char* what() const noexcept;
       };
     }
1 The class exception defines the base class for the types of objects thrown as exceptions by C++ standard
   library components, and certain expressions, to report errors detected during program execution.
<sup>2</sup> Each standard library class T that derives from class exception shall have a publicly accessible copy con-
   structor and a publicly accessible copy assignment operator that do not exit with an exception. These
   member functions shall meet the following postcondition: If two objects lhs and rhs both have dynamic
   type T and lhs is a copy of rhs, then strcmp(lhs.what(), rhs.what()) shall equal 0.
   exception() noexcept;
         Effects: Constructs an object of class exception.
         Remarks: Does not throw any exceptions.
   exception(const exception& rhs) noexcept;
   exception& operator=(const exception& rhs) noexcept;
         Effects: Copies an exception object.
        Postcondition: If *this and rhs both have dynamic type exception then strcmp(what(), rhs.what())
        shall equal 0.
   virtual ~exception();
        Effects: Destroys an object of class exception.
         Remarks: Does not throw any exceptions.
   virtual const char* what() const noexcept;
         Returns: An implementation-defined NTBS.
        Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion
        and display as a wstring (21.3, 22.4.1.4). The return value remains valid until the exception object
        from which it is obtained is destroyed or a non-const member function of the exception object is called.
```

448

```
[bad.exception]
   18.8.2 Class bad_exception
     namespace std {
       class bad_exception : public exception {
       public:
         bad_exception() noexcept;
         bad_exception(const bad_exception&) noexcept;
         bad_exception& operator=(const bad_exception&) noexcept;
         virtual const char* what() const noexcept;
       };
     }
  The class bad_exception defines the type of objects thrown as described in (15.5.2).
   bad_exception() noexcept;
2
        Effects: Constructs an object of class bad_exception.
3
        Remarks: The result of calling what () on the newly constructed object is implementation-defined.
       bad_exception(const bad_exception&) noexcept;
       bad_exception& operator=(const bad_exception&) noexcept;
        Effects: Copies an object of class bad_exception.
   virtual const char* what() const noexcept;
5
        Returns: An implementation-defined NTBS.
6
        Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion
        and display as a wstring (21.3, 22.4.1.4).
            Abnormal termination
                                                                                [exception.terminate]
   18.8.3
   18.8.3.1
             Type terminate_handler
                                                                                    [terminate.handler]
   typedef void (*terminate_handler)();
1
        The type of a handler function to be called by std::terminate() when terminating exception pro-
2
        Required behavior: A terminate_handler shall terminate execution of the program without returning
        to the caller.
3
        Default behavior: The implementation's default terminate_handler calls abort().
   18.8.3.2 set_terminate
                                                                                         [set.terminate]
   terminate_handler set_terminate(terminate_handler f) noexcept;
1
        Effects: Establishes the function designated by f as the current handler function for terminating
        exception processing.
2
        Remarks: It is unspecified whether a null pointer value designates the default terminate_handler.
3
        Returns: The previous terminate handler.
                                                                                         [get.terminate]
   18.8.3.3
             get_terminate
   terminate_handler get_terminate() noexcept;
1
        Returns: The current terminate_handler. [Note: This may be a null pointer value. — end note]
   § 18.8.3.3
                                                                                                      449
```

18.8.3.4 terminate [terminate]

[[noreturn]] void terminate() noexcept;

Remarks: Called by the implementation when exception handling must be abandoned for any of several reasons (15.5.1), in effect immediately after evaluating the throw-expression (18.8.3.1). May also be called directly by the program.

2 Effects: Calls the current terminate\_handler function. [Note: A default terminate\_handler is always considered a callable handler in this context. —end note]

#### 18.8.4 uncaught\_exception

1

[uncaught]

bool uncaught\_exception() noexcept;

Returns: true after the current thread has initialized an exception object (15.1) until a handler for the exception (including std::unexpected() or std::terminate()) is activated (15.3). [Note: This includes stack unwinding (15.2). —end note]

Remarks: When uncaught\_exception() returns true, throwing an exception can result in a call of std::terminate() (15.5.1).

# 18.8.5 Exception propagation

[propagation]

typedef unspecified exception\_ptr;

- The type exception\_ptr can be used to refer to an exception object.
- <sup>2</sup> exception\_ptr shall satisfy the requirements of NullablePointer (17.6.3.3).
- Two non-null values of type exception\_ptr are equivalent and compare equal if and only if they refer to the same exception.
- The default constructor of exception\_ptr produces the null value of the type.
- <sup>5</sup> exception\_ptr shall not be implicitly convertible to any arithmetic, enumeration, or pointer type.
- [ Note: An implementation might use a reference-counted smart pointer as exception\_ptr. -end note ]
- For purposes of determining the presence of a data race, operations on exception\_ptr objects shall access and modify only the exception\_ptr objects themselves and not the exceptions they refer to. Use of rethrow\_exception on exception\_ptr objects that refer to the same exception object shall not introduce a data race. [Note: if rethrow\_exception rethrows the same exception object (rather than a copy), concurrent access to that rethrown exception object may introduce a data race. Changes in the number of exception\_ptr objects that refer to a particular exception do not introduce a data race. end note]

exception\_ptr current\_exception() noexcept;

Returns: An exception\_ptr object that refers to the currently handled exception (15.3) or a copy of the currently handled exception, or a null exception\_ptr object if no exception is being handled. The referenced object shall remain valid at least as long as there is an exception\_ptr object that refers to it. If the function needs to allocate memory and the attempt fails, it returns an exception\_ptr object that refers to an instance of bad\_alloc. It is unspecified whether the return values of two successive calls to current\_exception refer to the same exception object. [Note: That is, it is unspecified whether current\_exception creates a new copy each time it is called. —end note] If the attempt to copy the current exception object throws an exception, the function returns an exception\_ptr object that refers to the thrown exception or, if this is not possible, to an instance of bad\_exception.

§ 18.8.5

[ Note: The copy constructor of the thrown exception may also fail, so the implementation is allowed to substitute a bad\_exception object to avoid infinite recursion. —  $end\ note$  ]

```
[[noreturn]] void rethrow_exception(exception_ptr p);
9
         Requires: p shall not be a null pointer.
10
         Throws: the exception object to which p refers.
   template<class E> exception_ptr make_exception_ptr(E e) noexcept;
11
         Effects: Creates an exception_ptr object that refers to a copy of e, as if
             throw e;
           } catch(...) {
             return current_exception();
12
         [Note: This function is provided for convenience and efficiency reasons. — end note]
                                                                                         [except.nested]
   18.8.6 nested_exception
     namespace std {
        class nested_exception {
        public:
          nested_exception() noexcept;
          nested_exception(const nested_exception&) noexcept = default;
          nested_exception& operator=(const nested_exception&) noexcept = default;
          virtual ~nested_exception() = default;
          // access functions
          [[noreturn]] void rethrow_nested() const;
          exception_ptr nested_ptr() const noexcept;
        [[noreturn]] template<class T> void throw_with_nested(T&& t);
        template <class E> void rethrow_if_nested(const E& e);
   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.
   [Note: nested_exception has a virtual destructor to make it a polymorphic class. Its presence can be
   tested for with dynamic_cast. — end note]
   nested_exception() noexcept;
3
         Effects: The constructor calls current_exception() and stores the returned value.
    [[noreturn]] void rethrow_nested() const;
         Effects: If nested_ptr() returns a null pointer, the function calls std::terminate(). Otherwise, it
         throws the stored exception captured by *this.
   exception_ptr nested_ptr() const noexcept;
   § 18.8.6
                                                                                                       451
```

5 Returns: The stored exception captured by this nested\_exception object.

```
[[noreturn]] template <class T> void throw_with_nested(T&& t);
Let U be remove_reference<T>::type.
```

- 6 Requires: U shall be CopyConstructible.
- 7 Throws: if U is a non-union class type not derived from nested\_exception, an exception of unspecified type that is publicly derived from both U and nested\_exception and constructed from std::forward<T>(t), otherwise std::forward<T>(t).

```
template <class E> void rethrow_if_nested(const E& e);
```

Effects: If the dynamic type of e is publicly and unambiguously derived from nested\_exception, calls dynamic\_cast<const nested\_exception&>(e).rethrow\_nested().

# 18.9 Initializer lists

[support.initlist]

The header <initializer\_list> defines one type.

```
Header <initializer_list> synopsis
```

```
namespace std {
  template<class E> class initializer_list {
  public:
    typedef E value_type;
    typedef const E& reference;
    typedef const E& const_reference;
    typedef size_t size_type;
    typedef const E* iterator;
    typedef const E* const_iterator;
    initializer_list() noexcept;
                                        // number of elements
    size_t size() const noexcept;
                                        // first element
    const E* begin() const noexcept;
    const E* end() const noexcept;
                                        // one past the last element
  // 18.9.3 initializer list range access
  template<class E> const E* begin(initializer_list<E> il) noexcept;
  template<class E> const E* end(initializer_list<E> il) noexcept;
```

An object of type initializer\_list<E> provides access to an array of objects of type const E. [Note: A pair of pointers or a pointer plus a length would be obvious representations for initializer\_list. initializer\_list is used to implement initializer lists as specified in 8.5.4. Copying an initializer list does not copy the underlying elements. —end note]

# 18.9.1 Initializer list constructors

[support.initlist.cons]

```
initializer_list() noexcept;

Effects: constructs an empty initializer_list object.
Postcondition: size() == 0
```

§ 18.9.1 452

#### 18.9.2 Initializer list access

[support.initlist.access]

const E\* begin() const noexcept;

1

1

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.

```
const E* end() const noexcept;
    Returns: begin() + size()
size_t size() const noexcept;
```

- 3 Returns: The number of elements in the array.
- 4 Complexity: constant time.

# 18.9.3 Initializer list range access

[support.initlist.range]

```
template<class E> const E* begin(initializer_list<E> il) noexcept;
    Returns: il.begin().

template<class E> const E* end(initializer_list<E> il) noexcept;
```

2 Returns: il.end().

# 18.10 Other runtime support

[support.runtime]

- Headers <csetjmp> (nonlocal jumps), <csignal> (signal handling), <cstdalign> (alignment), <cstdarg> (variable arguments), <cstdbool> (\_\_bool\_true\_false\_are\_defined). <cstdlib> (runtime environment getenv(), system()), and <ctime> (system clock clock(), time()) provide further compatibility with C code.
- The contents of these headers are the same as the Standard C library headers <setjmp.h>, <signal.h>, <stdalign.h>, <stdarg.h>, <stdbool.h>, <stdlib.h>, and <time.h>, respectively, with the following changes:
- The restrictions that ISO C places on the second parameter to the va\_start() macro in header <stdarg.h> are different in this International Standard. The parameter parmN is the identifier of the rightmost parameter in the variable parameter list of the function definition (the one just before the ...). 227 If the parameter parmN is 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.

<sup>4</sup> 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.

- <sup>5</sup> Calls to the function getenv shall not introduce a data race (17.6.5.9) provided that nothing modifies the environment. [Note: Calls to the POSIX functions setenv and putenv modify the environment. end note]
- A call to the setlocale function may introduce a data race with other calls to the setlocale function or with calls to functions that are affected by the current C locale. The implementation shall behave as if no library function other than locale::global() calls the setlocale function.

§ 18.10 453

<sup>227)</sup> Note that va\_start is required to work as specified even if unary operator& is overloaded for the type of parmN.

- 7 The header <cstdalign> and the header <stdalign.h> shall not define a macro named alignas.
- 8 The header <cstdbool> and the header <stdbool.h> shall not define macros named bool, true, or false.
- The common subset of the C and C++ languages consists of all declarations, definitions, and expressions that may appear in a well formed C++ program and also in a conforming C program. A POF ("plain old function") is a function that uses only features from this common subset, and that does not directly or indirectly use any function that is not a POF, except that it may use functions defined in Clause 29 that are not member functions. All signal handlers shall have C linkage. A POF that could be used as a signal handler in a conforming C program does not produce undefined behavior when used as a signal handler in a C++ program. The behavior of any other function used as a signal handler in a C++ program is implementation-defined.<sup>228</sup>

Table 34 — Header <csetjmp> synopsis

Type		Name(s)
Macro:	setjmp	
Type:	jmp_buf	
Function:	longjmp	

Table 35 — Header <csignal> synopsis

Type	Name(s)			
Macros:	SIGABRT	SIGILL	SIGSEGV	SIG_DFL
SIG_IGN	SIGFPE	SIGINT	SIGTERM	SIG_ERR
Type:	sig_atomic_t			
Functions:	raise	signal		

Table 36 — Header <cstdalign> synopsis

Type	$\mathbf{Name}(\mathbf{s})$	
Macro:	alignas_is_defined	

Table 37 — Header <cstdarg> synopsis

Type		Name(s)		
Macros:	va_arg	va_end	va_start	
va_copy				
Type:	va_list	<u> </u>		

§ 18.10 454

<sup>228)</sup> 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).

 $\odot$  ISO/IEC N3337

Table 38 — Header <cstdbool> synopsis

Type	Name(s)	
Macro:	bool_true_false_are_defined	

Table 39 — Header <cstdlib> synopsis

Type	Name(s)		
Functions:	getenv	system	

Table 40 — Header <ctime> synopsis

Type	Name(s)	
Macro:	CLOCKS_PER_SEC	
Type:	clock_t	
Function:	clock	

§ 18.10 455

# 19 Diagnostics library

# [diagnostics]

#### 19.1 General

[diagnostics.general]

- This Clause describes components that C++ programs may use to detect and report error conditions.
- <sup>2</sup> The following subclauses describe components for reporting several kinds of exceptional conditions, documenting program assertions, and a global variable for error number codes, as summarized in Table 41.

Table 41 — Diagnostics library summary

	Subclause	Header(s)
19.2	Exception classes	<stdexcept></stdexcept>
19.3	Assertions	<cassert></cassert>
19.4	Error numbers	<cerrno></cerrno>
19.5	System error support	<system_error></system_error>

## 19.2 Exception classes

[std.exceptions]

- The Standard C++ library provides classes to be used to report certain errors (17.6.5.12) in C++ programs. In the error model reflected in these classes, errors are divided into two broad categories: *logic* errors and *runtime* errors.
- <sup>2</sup> 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.
- <sup>3</sup> 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

```
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

[logic.error]

```
namespace std {
  class logic_error : public exception {
  public:
    explicit logic_error(const string& what_arg);
    explicit logic_error(const char* what_arg);
  };
}
```

§ 19.2.1 456

The class logic\_error defines the type of objects thrown as exceptions to report errors presumably de-

```
tectable before the program executes, such as violations of logical preconditions or class invariants.
   logic_error(const string& what_arg);
2
        Effects: Constructs an object of class logic error.
3
        Postcondition: strcmp(what(), what_arg.c_str()) == 0.
   logic_error(const char* what_arg);
4
        Effects: Constructs an object of class logic_error.
5
        Postcondition: strcmp(what(), what_arg) == 0.
   19.2.2
            Class domain error
                                                                                        [domain.error]
     namespace std {
       class domain_error : public logic_error {
       public:
         explicit domain_error(const string& what_arg);
         explicit domain_error(const char* what_arg);
     }
  The class domain_error defines the type of objects thrown as exceptions by the implementation to report
   domain errors.
   domain_error(const string& what_arg);
2
        Effects: Constructs an object of class domain_error.
3
        Postcondition: strcmp(what(), what arg.c str()) == 0.
   domain_error(const char* what_arg);
4
        Effects: Constructs an object of class domain_error.
5
        Postcondition: strcmp(what(), what arg) == 0.
   19.2.3
           Class invalid_argument
                                                                                   [invalid.argument]
     namespace std {
       class invalid_argument : public logic_error {
         explicit invalid_argument(const string& what_arg);
         explicit invalid_argument(const char* what_arg);
     }
  The class invalid_argument defines the type of objects thrown as exceptions to report an invalid argument.
   invalid_argument(const string& what_arg);
2
        Effects: Constructs an object of class invalid argument.
3
        Postcondition: strcmp(what(), what arg.c str()) == 0.
   invalid_argument(const char* what_arg);
4
        Effects: Constructs an object of class invalid_argument.
        Postcondition: strcmp(what(), what_arg) == 0.
   § 19.2.3
                                                                                                     457
```

```
[length.error]
   19.2.4 Class length_error
     namespace std {
       class length_error : public logic_error {
       public:
         explicit length_error(const string& what_arg);
         explicit length_error(const char* what_arg);
       };
     }
  The class length_error defines the type of objects thrown as exceptions to report an attempt to produce
   an object whose length exceeds its maximum allowable size.
   length_error(const string& what_arg);
2
        Effects: Constructs an object of class length error.
3
        Postcondition: strcmp(what(), what_arg.c_str()) == 0.
   length_error(const char* what_arg);
4
        Effects: Constructs an object of class length_error.
5
        Postcondition: strcmp(what(), what_arg) == 0.
                                                                                         [out.of.range]
   19.2.5
            Class out_of_range
     namespace std {
       class out_of_range : public logic_error {
         explicit out_of_range(const string& what_arg);
         explicit out_of_range(const char* what_arg);
       };
     }
<sup>1</sup> The class out_of_range defines the type of objects thrown as exceptions to report an argument value not
   in its expected range.
   out_of_range(const string& what_arg);
2
        Effects: Constructs an object of class out_of_range.
3
        Postcondition: strcmp(what(), what_arg.c_str()) == 0.
   out_of_range(const char* what_arg);
4
        Effects: Constructs an object of class out_of_range.
5
        Postcondition: strcmp(what(), what_arg) == 0.
                                                                                       [runtime.error]
   19.2.6
           Class runtime error
     namespace std {
       class runtime_error : public exception {
         explicit runtime_error(const string& what_arg);
         explicit runtime_error(const char* what_arg);
       };
     }
   § 19.2.6
                                                                                                     458
```

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.
           Class range_error
                                                                                          [range.error]
   19.2.7
     namespace std {
       class range_error : public runtime_error {
       public:
         explicit range_error(const string& what_arg);
         explicit range_error(const char* what_arg);
       };
     }
  The class range_error defines the type of objects thrown as exceptions to report range errors in internal
   computations.
   range_error(const string& what_arg);
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.
                                                                                      [overflow.error]
           Class overflow_error
   19.2.8
     namespace std {
       class overflow_error : public runtime_error {
       public:
         explicit overflow_error(const string& what_arg);
         explicit overflow_error(const char* what_arg);
       };
     }
  The class overflow_error defines the type of objects thrown as exceptions to report an arithmetic overflow
   error.
   overflow_error(const string& what_arg);
2
        Effects: Constructs an object of class overflow_error.
3
        Postcondition: strcmp(what(), what_arg.c_str()) == 0.
   overflow_error(const char* what_arg);
4
        Effects: Constructs an object of class overflow_error.
        Postcondition: strcmp(what(), what_arg) == 0.
   § 19.2.8
                                                                                                     459
```

## 19.2.9 Class underflow\_error

[underflow.error]

```
namespace std {
  class underflow_error : public runtime_error {
  public:
    explicit underflow_error(const string& what_arg);
    explicit underflow_error(const char* what_arg);
  };
}
```

The class underflow\_error defines the type of objects thrown as exceptions to report an arithmetic underflow error.

```
underflow_error(const string& what_arg);

Effects: Constructs an object of class underflow_error.

Postcondition: strcmp(what(), what_arg.c_str()) == 0.

underflow_error(const char* what_arg);

Effects: Constructs an object of class underflow_error.

Postcondition: strcmp(what(), what_arg) == 0.
```

19.3 Assertions [assertions]

<sup>1</sup> The header <cassert>, described in (Table 42), provides a macro for documenting C++ program assertions and a mechanism for disabling the assertion checks.

Table 42 — Header <cassert> synopsis

Type	Name(s)
Macro:	assert

The contents are the same as the Standard C library header <assert.h>. SEE ALSO: ISO C 7.2.

#### 19.4 Error numbers

[errno]

The header <cerrno> is described in Table 43. Its contents are the same as the POSIX header <errno.h>, except that errno shall be defined as a macro. [Note: The intent is to remain in close alignment with the POSIX standard. —end note] A separate errno value shall be provided for each thread.

#### 19.5 System error support

[syserr]

- <sup>1</sup> 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.
- <sup>2</sup> Components described in this subclause shall not change the value of errno (19.4). Implementations should leave the error states provided by other libraries unchanged.

Header <system\_error> synopsis

```
namespace std {
  class error_category;
  const error_category& generic_category() noexcept;
  const error_category& system_category() noexcept;
  class error_code;
```

§ 19.5

Table 43 — Header <cerrno> synopsis

Type			Name(s)		
Macros:	ECONNREFUSED	EIO	ENODEV	ENOTEMPTY	ERANGE
E2BIG	ECONNRESET	EISCONN	ENOENT	ENOTRECOVERABLE	EROFS
EACCES	EDEADLK	EISDIR	ENOEXEC	ENOTSOCK	ESPIPE
EADDRINUSE	EDESTADDRREQ	EL00P	ENOLCK	ENOTSUP	ESRCH
EADDRNOTAVAIL	EDOM	EMFILE	ENOLINK	ENOTTY	ETIME
EAFNOSUPPORT	EEXIST	EMLINK	ENOMEM	ENXIO	ETIMEDOUT
EAGAIN	EFAULT	EMSGSIZE	ENOMSG	EOPNOTSUPP	ETXTBSY
EALREADY	EFBIG	ENAMETOOLONG	ENOPROTOOPT	EOVERFLOW	EWOULDBLOCK
EBADF	EHOSTUNREACH	ENETDOWN	ENOSPC	EOWNERDEAD	EXDEV
EBADMSG	EIDRM	ENETRESET	ENOSR	EPERM	errno
EBUSY	EILSEQ	ENETUNREACH	ENOSTR	EPIPE	
ECANCELED	EINPROGRESS	ENFILE	ENOSYS	EPROTO	
ECHILD	EINTR	ENOBUFS	ENOTCONN	EPROTONOSUPPORT	
ECONNABORTED	EINVAL	ENODATA	ENOTDIR	EPROTOTYPE	

```
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 {
                                      // EAFNOSUPPORT
  address_family_not_supported,
                                      // EADDRINUSE
  address_in_use,
                                      // EADDRNOTAVAIL
  address_not_available,
                                      // EISCONN
  already_connected,
                                      // E2BIG
  argument_list_too_long,
                                      // EDOM
  argument_out_of_domain,
                                      // EFAULT
  bad_address,
                                      // EBADF
  bad_file_descriptor,
                                      // EBADMSG
  bad_message,
                                      // EPIPE
  broken_pipe,
                                      // ECONNABORTED
  connection_aborted,
                                      // EALREADY
  connection_already_in_progress,
                                      // ECONNREFUSED
  connection_refused,
                                      // ECONNRESET
  connection_reset,
                                      // EXDEV
  cross_device_link,
                                      // EDESTADDRREQ
  destination_address_required,
                                       // EBUSY
  device_or_resource_busy,
                                       // ENOTEMPTY
  directory_not_empty,
                                      // ENOEXEC
  executable_format_error,
                                      // EEXIST
  file_exists,
                                      // EFBIG
  file_too_large,
                                      // ENAMETOOLONG
  filename_too_long,
                                      // ENOSYS
  function_not_supported,
```

§ 19.5

 $\odot$  ISO/IEC N3337

host_unreachable,	// EHOSTUNREACH
identifier_removed,	// EIDRM
illegal_byte_sequence,	// EILSEQ
<pre>inappropriate_io_control_operation,</pre>	// ENOTTY
interrupted,	// EINTR
invalid_argument,	// EINVAL
invalid_seek,	// ESPIPE
io_error,	// EIO
is_a_directory,	// EISDIR
message_size,	// EMSGSIZE
network_down,	// ENETDOWN
network_reset,	// ENETRESET
network_unreachable,	// ENETUNREACH
no_buffer_space,	// ENOBUFS
no_child_process,	// ECHILD
no_link,	// ENOLINK
no_lock_available,	// ENOLCK
no_message_available,	// ENODATA
no_message,	// ENOMSG
no_protocol_option,	// ENOPROTOOPT
no_space_on_device,	// ENOSPC
no_stream_resources,	// ENOSR
no_such_device_or_address,	// ENXIO
no_such_device,	// ENODEV
no_such_file_or_directory,	// ENOENT
no_such_process,	// ESRCH
not_a_directory,	// ENOTDIR
not_a_socket,	// ENOTSOCK
not_a_stream,	// ENOSTR
not_connected,	// ENOTCONN
not_enough_memory,	// ENOMEM
	// ENOTSUP
not_supported,	// ECANCELED
operation_canceled,	// EINPROGRESS
operation_in_progress,	// EPERM
operation_not_permitted,	' ( .
operation_not_supported,	// EUOUL DRI OCK
operation_would_block,	// EWOULDBLOCK
owner_dead,	// EOWNERDEAD
permission_denied,	// EACCES
protocol_error,	// EPROTO
protocol_not_supported,	// EPROTONOSUPPORT
read_only_file_system,	// EROFS
resource_deadlock_would_occur,	// EDEADLK
resource_unavailable_try_again,	// EAGAIN
result_out_of_range,	// ERANGE
state_not_recoverable,	// ENOTRECOVERABLE
stream_timeout,	// ETIME
text_file_busy,	// ETXTBSY
timed_out,	// ETIMEDOUT
too_many_files_open_in_system,	// ENFILE
too_many_files_open,	// EMFILE
too_many_links,	// EMLINK
too_many_symbolic_link_levels,	// ELOOP
<pre>value_too_large,</pre>	// EOVERFLOW
wrong_protocol_type,	// EPROTOTYPE

§ 19.5

```
};
  template <> struct is_error_condition_enum<errc> : true_type { }
  error_code make_error_code(errc e) noexcept;
  error_condition make_error_condition(errc e) noexcept;
  // 19.5.4 Comparison operators:
  bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
  bool operator == (const error_code& lhs, const error_condition& rhs) noexcept;
  bool operator == (const error_condition& lhs, const error_code& rhs) noexcept;
  bool operator == (const error_condition& lhs, const error_condition& rhs) noexcept;
  bool operator!=(const error_code& lhs, const error_code& rhs) noexcept;
  bool operator!=(const error_code& lhs, const error_condition& rhs) noexcept;
  bool operator!=(const error_condition& lhs, const error_code& rhs) noexcept;
  bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;
  // 19.5.5 Hash support
 template <class T> struct hash;
  template <> struct hash<error_code>;
} // namespace std
```

- 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.
- <sup>4</sup> The is\_error\_code\_enum and is\_error\_condition\_enum may be specialized for user-defined types to indicate that such types are eligible for class error\_code and class error\_condition automatic conversions, respectively.

### 19.5.1 Class error\_category

[syserr.errcat]

## 19.5.1.1 Class error\_category overview

[syserr.errcat.overview]

The class error\_category serves as a base class for types used to identify the source and encoding of a particular category of error code. Classes may be derived from error\_category to support categories of errors in addition to those defined in this International Standard. Such classes shall behave as specified in this subclause. [Note: error\_category objects are passed by reference, and two such objects are equal if they have the same address. This means that applications using custom error\_category types should create a single object of each such type. —end note]

```
namespace std {
  class error_category {
  public:
    virtual ~error_category() noexcept;
    error_category(const error_category&) = delete;
    error_category& operator=(const error_category&) = delete;
    virtual const char* name() const noexcept = 0;
    virtual error_condition default_error_condition(int ev) const noexcept;
    virtual bool equivalent(int code, const error_condition& condition) const noexcept;
    virtual bool equivalent(const error_code& code, int condition) const noexcept;
    virtual string message(int ev) const = 0;

  bool operator==(const error_category& rhs) const noexcept;
  bool operator!=(const error_category& rhs) const noexcept;
  bool operator<(const error_category& rhs) const noexcept;
  bool operator<(const error_category& rhs) const noexcept;
};</pre>
```

§ 19.5.1.1 463

```
const error_category& generic_category() noexcept;
       const error_category& system_category() noexcept;
       // namespace std
   19.5.1.2 Class error_category virtual members
                                                                               [syserr.errcat.virtuals]
   virtual const char* name() const noexcept = 0;
        Returns: A string naming the error category.
   virtual error_condition default_error_condition(int ev) const noexcept;
        Returns: error_condition(ev, *this).
   virtual bool equivalent(int code, const error_condition& condition) const noexcept;
        Returns: default_error_condition(code) == condition.
   virtual bool equivalent(const error_code& code, int condition) const noexcept;
        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
                                                                           [syserr.errcat.nonvirtuals]
   bool operator==(const error_category& rhs) const noexcept;
1
        Returns: this == &rhs.
   bool operator!=(const error_category& rhs) const noexcept;
        Returns: !(*this == rhs).
   bool operator<(const error_category& rhs) const noexcept;</pre>
3
        Returns: less<const error category*>()(this, &rhs).
        [Note: less (20.8.5) provides a total ordering for pointers. — end note]
   19.5.1.4 Program defined classes derived from error_category
                                                                               [syserr.errcat.derived]
   virtual const char *name() const noexcept = 0;
        Returns: A string naming the error category.
   virtual error_condition default_error_condition(int ev) const noexcept;
        Returns: An object of type error_condition that corresponds to ev.
```

§ 19.5.1.4 464

virtual bool equivalent(int code, const error\_condition% condition) const noexcept;

Returns: true if, for the category of error represented by \*this, code is considered equivalent to condition; otherwise, false.

virtual bool equivalent(const error\_code& code, int condition) const noexcept;

4 Returns: true if, for the category of error represented by \*this, code is considered equivalent to condition; otherwise, false.

#### 19.5.1.5 Error category objects

1

[syserr.errcat.objects]

const error\_category& generic\_category() noexcept;

- Returns: A reference to an object of a type derived from class error\_category. All calls to this function shall return references to the same object.
- Remarks: The object's default\_error\_condition and equivalent virtual functions shall behave as specified for the class error\_category. The object's name virtual function shall return a pointer to the string "generic".

const error\_category& system\_category() noexcept;

- Returns: A reference to an object of a type derived from class error\_category. All calls to this function shall return references to the same object.
- 4 Remarks: The object's equivalent virtual functions shall behave as specified for class error\_category. The object's name virtual function shall return a pointer to the string "system". The object's default\_error\_condition virtual function shall behave as follows:

If the argument ev corresponds to a POSIX errno value posv, the function shall return error\_condition(posv, generic\_category()). Otherwise, the function shall return error\_condition(ev, system\_category()). What constitutes correspondence for any given operating system is unspecified. [Note: The number of potential system error codes is large and unbounded, and some may not correspond to any POSIX errno value. Thus implementations are given latitude in determining correspondence. — end note]

## 19.5.2 Class error\_code

[syserr.errcode]

## 19.5.2.1 Class error\_code overview

[syserr.errcode.overview]

The class error\_code describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces. [Note: Class error\_code is an adjunct to error reporting by exception. —end note]

§ 19.5.2.1 465

```
template <class ErrorCodeEnum>
          error_code& operator=(ErrorCodeEnum e) noexcept;
      void clear() noexcept;
      // 19.5.2.4 observers:
      int value() const noexcept;
      const error_category& category() const noexcept;
      error_condition default_error_condition() const noexcept;
      string message() const;
      explicit operator bool() const noexcept;
   private:
      int val_;
                                  // exposition only
      const error_category* cat_; // exposition only
    };
    // 19.5.2.5 non-member functions:
    error_code make_error_code(errc e) noexcept;
    bool operator<(const error_code& lhs, const error_code& rhs) noexcept;</pre>
    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
                                                                      [syserr.errcode.constructors]
error_code() noexcept;
     Effects: Constructs an object of type error_code.
     Postconditions: val_ == 0 and cat_ == &system_category().
error_code(int val, const error_category& cat) noexcept;
     Effects: Constructs an object of type error_code.
     Postconditions: val_ == val and cat_ == &cat.
template <class ErrorCodeEnum>
  error_code(ErrorCodeEnum e) noexcept;
     Effects: Constructs an object of type error_code.
     Postconditions: *this == make_error_code(e).
                   This constructor shall not participate
                                                                       overload resolution
                                                                                               unless
                                                                   _{
m in}
     is_error_code_enum<ErrorCodeEnum>::value is true.
19.5.2.3 Class error_code modifiers
                                                                         [syserr.errcode.modifiers]
void assign(int val, const error_category& cat) noexcept;
     Postconditions: val_ == val and cat_ == &cat.
template <class ErrorCodeEnum>
   error_code& operator=(ErrorCodeEnum e) noexcept;
§ 19.5.2.3
                                                                                                  466
```

1

3

5

6

```
2
        Postconditions: *this == make_error_code(e).
3
        Returns: *this.
4
        Remarks:
                      This
                                        shall
                                                                        overload
                                                                                   resolution
                                                                                               unless
                             operator
                                               not
                                                      participate
        is_error_code_enum<ErrorCodeEnum>::value is true.
  void clear() noexcept;
        Postconditions: value() == 0 and category() == system_category().
  19.5.2.4 Class error_code observers
                                                                          [syserr.errcode.observers]
  int value() const noexcept;
        Returns: val .
  const error_category& category() const noexcept;
2
        Returns: *cat_.
  error_condition default_error_condition() const noexcept;
3
        Returns: category().default_error_condition(value()).
  string message() const;
        Returns: category().message(value()).
  explicit operator bool() const noexcept;
        Returns: value() != 0.
  19.5.2.5 Class error_code non-member functions
                                                                       [syserr.errcode.nonmembers]
  error_code make_error_code(errc e) noexcept;
        Returns: error_code(static_cast<int>(e), generic_category()).
  bool operator<(const error_code& lhs, const error_code& rhs) noexcept;</pre>
2
        Returns: lhs.category() < rhs.category() || lhs.category() == rhs.category() && lhs.value()
        < rhs.value().
  template <class charT, class traits>
    basic_ostream<charT,traits>&
      operator<<(basic_ostream<charT,traits>& os, const error_code& ec);
3
        Effects: os << ec.category().name() << ':' << ec.value().</pre>
```

§ 19.5.2.5

#### 19.5.3 Class error\_condition

1

3

5

[syserr.errcondition]

#### 19.5.3.1 Class error\_condition overview

[syserr.errcondition.overview]

The class error\_condition describes an object used to hold values identifying error conditions. [Note: error\_condition values are portable abstractions, while error\_code values (19.5.2) are implementation specific. —end note]

```
namespace std {
   class error_condition {
    public:
      // 19.5.3.2 constructors:
      error_condition() noexcept;
      error_condition(int val, const error_category& cat) noexcept;
      template <class ErrorConditionEnum>
        error_condition(ErrorConditionEnum e) noexcept;
      // 19.5.3.3 modifiers:
      void assign(int val, const error_category& cat) noexcept;
      template < class ErrorConditionEnum>
          error_condition& operator=(ErrorConditionEnum e) noexcept;
      void clear() noexcept;
      // 19.5.3.4 observers:
      int value() const noexcept;
      const error_category& category() const noexcept;
      string message() const;
      explicit operator bool() const noexcept;
    private:
      int val_;
                                  // exposition only
      const error_category* cat_; // exposition only
    // 19.5.3.5 non-member functions:
   bool operator<(const error_condition& lhs, const error_condition& rhs) noexcept;
 } // namespace std
19.5.3.2 Class error_condition constructors
                                                                  [syserr.errcondition.constructors]
error_condition() noexcept;
     Effects: Constructs an object of type error_condition.
     Postconditions: val_ == 0 and cat_ == &generic_category().
error_condition(int val, const error_category& cat) noexcept;
     Effects: Constructs an object of type error_condition.
     Postconditions: val_ == val and cat_ == &cat.
template <class ErrorConditionEnum>
  error_condition(ErrorConditionEnum e) noexcept;
     Effects: Constructs an object of type error_condition.
     Postcondition: *this == make_error_condition(e).
```

§ 19.5.3.2 468

overload

resolution

unless

constructor shall not participate in

7

This

```
is_error_condition_enum<ErrorConditionEnum>::value is true.
  19.5.3.3 Class error_condition modifiers
                                                                     [syserr.errcondition.modifiers]
  void assign(int val, const error_category& cat) noexcept;
        Postconditions: val_ == val and cat_ == &cat.
  template <class ErrorConditionEnum>
       error_condition& operator=(ErrorConditionEnum e) noexcept;
2
        Postcondition: *this == make_error_condition(e).
3
        Returns: *this.
4
        Remarks:
                      This
                             operator
                                        _{\mathrm{shall}}
                                               not
                                                      participate
                                                                   in
                                                                        overload
                                                                                   resolution
                                                                                               unless
        is_error_condition_enum<ErrorConditionEnum>::value is true.
  void clear() noexcept;
        Postconditions: value() == 0 and category() == generic_category().
  19.5.3.4 Class error_condition observers
                                                                     [syserr.errcondition.observers]
  int value() const noexcept;
        Returns: val_.
  const error_category& category() const noexcept;
        Returns: *cat_.
  string message() const;
        Returns: category().message(value()).
  explicit operator bool() const noexcept;
        Returns: value() != 0.
  19.5.3.5 Class error_condition non-member functions
                                                                  [syserr.errcondition.nonmembers]
  error_condition make_error_condition(errc e) noexcept;
        Returns: error_condition(static_cast<int>(e), generic_category()).
  bool operator<(const error_condition& lhs, const error_condition& rhs) noexcept;
1
        Returns: lhs.category() < rhs.category() || lhs.category() == rhs.category() &&
        lhs.value() < rhs.value().</pre>
```

§ 19.5.3.5

```
19.5.4 Comparison operators
                                                                                   [syserr.compare]
   bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
        Returns: lhs.category() == rhs.category() && lhs.value() == rhs.value().
1
   bool operator == (const error_code& lhs, const error_condition& rhs) noexcept;
2
        Returns: lhs.category().equivalent(lhs.value(), rhs) || rhs.category().equivalent(lhs,
        rhs.value()).
   bool operator == (const error_condition& lhs, const error_code& rhs) noexcept;
3
        Returns: rhs.category().equivalent(rhs.value(), lhs) || lhs.category().equivalent(rhs,
        lhs.value()).
   bool operator == (const error_condition& lhs, const error_condition& rhs) noexcept;
        Returns: lhs.category() == rhs.category() && lhs.value() == rhs.value().
   bool operator!=(const error_code& lhs, const error_code& rhs) noexcept;
   bool operator!=(const error_code& lhs, const error_condition& rhs) noexcept;
   bool operator!=(const error_condition& lhs, const error_code& rhs) noexcept;
   bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;
        Returns: !(lhs == rhs).
   19.5.5 System error hash support
                                                                                       [syserr.hash]
   template <> struct hash<error_code>;
        Requires: the template specialization shall meet the requirements of class template hash (20.8.12).
   19.5.6 Class system_error
                                                                                      [syserr.syserr]
   19.5.6.1 Class system_error overview
                                                                             [syserr.syserr.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,
```

§ 19.5.6.1 470

const char\* what\_arg);

system\_error(int ev, const error\_category& ecat);

const error\_code& code() const noexcept;

```
const char* what() const noexcept;
       }:
          // namespace std
   19.5.6.2 Class system_error members
                                                                                [syserr.syserr.members]
   system_error(error_code ec, const string& what_arg);
1
         Effects: Constructs an object of class system_error.
2
         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.
         string(what()).find(what_arg) != string::npos.
   system_error(error_code ec);
5
         Effects: Constructs an object of class system_error.
6
         Postconditions: code() == ec.
   system_error(int ev, const error_category& ecat,
      const string& what_arg);
7
         Effects: Constructs an object of class system_error.
8
         Postconditions: code() == error_code(ev, ecat).
         string(what()).find(what_arg) != string::npos.
   system_error(int ev, const error_category& ecat,
      const char* what_arg);
9
         Effects: Constructs an object of class system_error.
10
         Postconditions: code() == error_code(ev, ecat).
         string(what()).find(what_arg) != string::npos.
   system_error(int ev, const error_category& ecat);
11
         Effects: Constructs an object of class system_error.
12
         Postconditions: code() == error_code(ev, ecat).
   const error_code& code() const noexcept;
13
         Returns: ec or error_code(ev, ecat), from the constructor, as appropriate.
   const char *what() const noexcept;
14
         Returns: An NTBS incorporating the arguments supplied in the constructor.
         [Note: The returned NTBS might be the contents of what_arg + ": " + code.message(). — end
         note
```

§ 19.5.6.2

471

# 20 General utilities library

# [utilities]

## 20.1 General [utilities.general]

This Clause describes utilities that are generally useful in C++ programs; some of these utilities are used by other elements of the C++ standard library. These utilities are summarized in Table 44.

	Subclause	Header(s)
20.2	Utility components	<utility></utility>
20.3	Pairs	<utility></utility>
20.4	Tuples	<tuple></tuple>
20.5	Fixed-size sequences of bits	   
		<memory></memory>
20.6	Memory	<cstdlib></cstdlib>
		<cstring></cstring>
20.7	Smart pointers	<memory></memory>
20.8	Function objects	<functional></functional>
20.9	Type traits	<type_traits></type_traits>
20.10	Compile-time rational arithmetic	<ratio></ratio>
20.11	Time utilities	<chrono></chrono>
		<ctime></ctime>
20.12	Scoped allocators	<pre><scoped_allocator></scoped_allocator></pre>
20.13	Type indexes	<typeindex></typeindex>

Table 44 — General utilities library summary

## 20.2 Utility components

[utility]

This subclause contains some basic function and class templates that are used throughout the rest of the library.

## Header <utility> synopsis

The header <utility> defines several types and function templates that are described in this Clause. It also defines the template pair and various function templates that operate on pair objects.

#include <initializer\_list>

```
namespace std {
   // 20.2.1, operators:
   namespace rel_ops {
     template<class T> bool operator!=(const T&, const T&);
     template<class T> bool operator> (const T&, const T&);
     template<class T> bool operator<=(const T&, const T&);
     template<class T> bool operator>=(const T&, const T&);
}

// 20.2.2, swap:
template<class T> void swap(T& a, T& b) noexcept(see below);
template <class T, size_t N> void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));
```

§ 20.2 472

```
// 20.2.3, forward/move:
template <class T> T&& forward(typename remove_reference<T>::type& t) noexcept;
template <class T> T&& forward(typename remove_reference<T>::type&& t) noexcept;
template <class T> typename remove_reference<T>::type&& move(T&&) noexcept;
template <class T> typename conditional<
  !is_nothrow_move_constructible<T>::value && is_copy_constructible<T>::value,
  const T&, T&&>::type move_if_noexcept(T& x) noexcept;
// 20.2.4, declval:
template <class T>
  typename add_rvalue_reference<T>::type declval() noexcept; // as unevaluated operand
// 20.3, pairs:
template <class T1, class T2> struct pair;
// 20.3.3, pair specialized algorithms:
template <class T1, class T2>
  bool operator == (const pair <T1, T2>&, const pair <T1, T2>&);
template <class T1, class T2>
  bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
  bool operator!=(const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
  bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
  bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
  bool operator <= (const pair <T1,T2>&, const pair <T1,T2>&);
template <class T1, class T2>
  void swap(pair<T1,T2>& x, pair<T1,T2>& y) noexcept(noexcept(x.swap(y)));
template <class T1, class T2>
  see below make_pair(T1&&, T2&&);
// 20.3.4, tuple-like access to pair:
template <class T> class tuple_size;
template <size_t I, class T> class tuple_element;
template <class T1, class T2> struct tuple_size<std::pair<T1, T2> >;
template <class T1, class T2> struct tuple_element<0, std::pair<T1, T2> >;
template <class T1, class T2> struct tuple_element<1, std::pair<T1, T2> >;
template<size_t I, class T1, class T2>
  typename tuple_element<I, std::pair<T1, T2> >::type& get(std::pair<T1, T2>&) noexcept;
template<size_t I, class T1, class T2>
  typename tuple_element<I, std::pair<T1, T2> >::type&& get(std::pair<T1, T2>&&) noexcept;
template<size_t I, class T1, class T2>
  const typename tuple_element<I, std::pair<T1, T2> >::type&
    get(const std::pair<T1, T2>&) noexcept;
// 20.3.5, pair piecewise construction
struct piecewise_construct_t { };
constexpr piecewise_construct_t piecewise_construct = piecewise_construct_t();
template <class... Types> class tuple; // defined in <tuple>
```

§ 20.2 473

```
20.2.1 Operators
                                                                                             [operators]
  To avoid redundant definitions of operator!= out of operator== and operators >, <=, and >= out of
   operator<, the library provides the following:
   template <class T> bool operator!=(const T& x, const T& y);
2
        Requires: Type T is EqualityComparable (Table 17).
3
        Returns: !(x == y).
   template <class T> bool operator>(const T& x, const T& y);
4
        Requires: Type T is LessThanComparable (Table 18).
5
        Returns: y < x.
   template <class T> bool operator<=(const T& x, const T& y);</pre>
6
        Requires: Type T is LessThanComparable (Table 18).
7
        Returns: !(y < x).
   template <class T> bool operator>=(const T& x, const T& y);
8
        Requires: Type T is LessThanComparable (Table 18).
9
        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.
                                                                                          [utility.swap]
   20.2.2
            swap
   template<class T> void swap(T& a, T& b) noexcept(see below);
1
        Remark: The expression inside noexcept is equivalent to:
          is_nothrow_move_constructible<T>::value &&
          is_nothrow_move_assignable<T>::value
2
        Requires: Type T shall be MoveConstructible (Table 20) and MoveAssignable (Table 22).
3
        Effects: Exchanges values stored in two locations.
   template<class T, size_t N>
     void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));
4
        Requires: a[i] shall be swappable with (17.6.3.2) b[i] for all i in the range [0, N).
5
        Effects: swap_ranges(a, a + N, b)
```

§ 20.2.2 474

## 20.2.3 forward/move helpers

[forward]

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> T&& forward(typename remove_reference<T>::type& t) noexcept;
   template <class T> T&& forward(typename remove_reference<T>::type&& t) noexcept;
2
        Returns: static_cast<T&&>(t).
3
        if the second form is instantiated with an Ivalue reference type, the program is ill-formed.
4
        [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
5
        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) noexcept;
        Returns: static_cast<typename remove_reference<T>::type&&>(t).
6
        [Example:
          template <class T, class A1>
          shared_ptr<T> factory(A1&& a1) {
            return shared_ptr<T>(new T(std::forward<A1>(a1)));
          }
          struct A {
            A();
            A(const A&); // copies from lvalues
                           // moves from rvalues
            A(A&&);
          };
          void g() {
            A a;
                                                              // "a" binds to A(const A&)
            shared_ptr<A> sp1 = factory<A>(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

§ 20.2.3 475

```
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<
!is_nothrow_move_constructible<T>::value && is_copy_constructible<T>::value,
const T&, T&&>::type move_if_noexcept(T& x) noexcept;
```

Returns: std::move(x)

## 20.2.4 Function template declval

[declval]

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

Remarks: If this function is odr-used (3.2), the program is ill-formed.

Remarks: The template parameter T of declval may be an incomplete type.

[Example:
    template <class To, class From>
    decltype(static_cast<To>(declval<From>())) convert(From&&);
```

declares a function template convert which only participates in overloading if the type From can be explicitly converted to type To. For another example see class template common\_type (20.9.7.6). — end example

20.3 Pairs [pairs]

### 20.3.1 In general

[pairs.general]

The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to pair objects as if they were tuple objects (see 20.4.2.5 and 20.4.2.6).

#### 20.3.2 Class template pair

[pairs.pair]

```
// defined in header <utility>
namespace std {
  template <class T1, class T2>
  struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair(const pair&) = default;
    pair(pair&&) = default;
    constexpr pair();
    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>
```

§ 20.3.2

pair(piecewise\_construct\_t,

```
tuple<Args1...> first_args, tuple<Args2...> second_args);
          pair& operator=(const pair& p);
          template<class U, class V> pair& operator=(const pair<U, V>& p);
          pair& operator=(pair&& p) noexcept(see below);
          template < class U, class V> pair& operator = (pair < U, V>&& p);
          void swap(pair& p) noexcept(see below);
       };
     }
   Constructors and member function of pair shall not throw exceptions unless one of the element-wise oper-
   ations specified to be called for that operation throws an exception.
    constexpr pair();
2
         Requires: is_default_constructible<first_type>::value is true and is_default_construct-
         ible<second_type>::value is true.
3
         Effects: Value-initializes first and second.
   pair(const T1& x, const T2& y);
4
         Requires: is_copy_constructible<first_type>::value is true and is_copy_constructible<sec-
         ond_type>::value is true.
         Effects: The constructor initializes first with x and second with y.
   template < class U, class V> pair(U&& x, V&& y);
6
         Requires: is_constructible<first_type, U&&>::value is true and is_constructible<second_-
         type, V&&>::value is true.
         Effects: The constructor initializes first with std::forward<U>(x) and second with std::forward<
         V>(y).
8
         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.
   template<class U, class V> pair(const pair<U, V>& p);
9
         Requires: is_constructible<first_type, const U&>::value is true and is_constructible<sec-
         ond_type, const V&>::value is true.
10
         Effects: Initializes members from the corresponding members of the argument.
11
         Remark: This constructor shall not participate in overload resolution unless const U& is implicitly
         convertible to first_type and const V& is implicitly convertible to second_type.
   template<class U, class V> pair(pair<U, V>&& p);
12
         Requires: is_constructible<first_type, U&&>::value is true and is_constructible<second_-
         type, V&&>::value is true.
13
         Effects: The constructor initializes first with std::forward<U>(p.first) and second with std::
         forward<V>(p.second).
14
         Remark: This constructor shall not participate in overload resolution unless U is implicitly convertible
         to first_type and V is implicitly convertible to second_type.
   § 20.3.2
```

477

```
template<class... Args1, class... Args2>
     pair(piecewise_construct_t,
          tuple<Args1...> first_args, tuple<Args2...> second_args);
15
         Requires: Requires: is_constructible<first_type, Args1&&...>::value is true and is_con-
         structible<second_type, Args2&&...>::value is true.
16
         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.
   pair& operator=(const pair& p);
17
         Requires: is_copy_assignable<first_type>::value is true and is_copy_assignable<second_-
         type>::value is true.
18
         Effects: Assigns p.first to first and p.second to second.
19
         Returns: *this.
   template<class U, class V> pair& operator=(const pair<U, V>& p);
20
         Requires: is_assignable<first_type&, const U&>::value is true and is_assignable<second type&,
         const V&>::value is true.
21
         Effects: Assigns p.first to first and p.second to second.
22
         Returns: *this.
   pair& operator=(pair&& p) noexcept(see below);
23
         Remarks: The expression inside noexcept is equivalent to:
           is_nothrow_move_assignable<T1>::value &&
           is_nothrow_move_assignable<T2>::value
24
         Requires: is_move_assignable<first_type>::value is true and is_move_assignable<second_-
         type>::value is true.
25
         Effects: Assigns to first with std::forward<first_type>(p.first) and to second with
         std::forward<second_type>(p.second).
26
         Returns: *this.
   template < class U, class V> pair& operator = (pair < U, V > && p);
27
         Requires: is_assignable<first_type&, U&&>::value is true and is_assignable<second_type&,
         V&&>::value is true.
28
         Effects: Assigns to first with std::forward<U>(p.first) and to second with
         std::forward<V>(p.second).
29
         Returns: *this.
```

§ 20.3.2 478

```
void swap(pair& p) noexcept(see below);
30
         Remarks: The expression inside noexcept is equivalent to:
           noexcept(swap(first, p.first)) &&
           noexcept(swap(second, p.second))
31
         Requires: first shall be swappable with (17.6.3.2) p.first and second shall be swappable with
32
         Effects: Swaps first with p.first and second with p.second.
            Specialized algorithms
                                                                                          [pairs.spec]
   template <class T1, class T2>
     bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);
         Returns: x.first == y.first && x.second == y.second.
   template <class T1, class T2>
     bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y);
         Returns: x.first < y.first || (!(y.first < x.first) && x.second < y.second).
   template <class T1, class T2>
     bool operator!=(const pair<T1, T2>& x, const pair<T1, T2>& y);
         Returns: !(x == y)
   template <class T1, class T2>
     bool operator>(const pair<T1, T2>& x, const pair<T1, T2>& y);
         Returns: y < x
   template <class T1, class T2>
     bool operator>=(const pair<T1, T2>& x, const pair<T1, T2>& y);
         Returns: !(x < y)
   template <class T1, class T2>
     bool operator<=(const pair<T1, T2>& x, const pair<T1, T2>& y);
         Returns: !(y < x)
   template<class T1, class T2> void swap(pair<T1, T2>& x, pair<T1, T2>& y)
     noexcept(noexcept(x.swap(y)));
         Effects: x.swap(y)
   template <class T1, class T2>
     pair<V1, V2> make_pair(T1&& x, T2&& y);
                                                                                                    479
   § 20.3.3
```

```
8
        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.
9
        [ 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]
   20.3.4
           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
        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>&) noexcept;
   template<size_t I, class T1, class T2>
     const typename tuple_element<I, std::pair<T1, T2> >::type& get(const pair<T1, T2>&) noexcept;
        Returns: If I == 0 returns p.first; if I == 1 returns p.second; otherwise the program is ill-formed.
   template<size_t I, class T1, class T2>
     typename tuple_element<I, std::pair<T1, T2> >::type&& get(std::pair<T1, T2>&&) noexcept;
6
        Returns: If I == 0 returns std::forward<T1&&>(p.first); if I == 1 returns std::forward<T2&&>(
        p.second); otherwise the program is ill-formed.
   20.3.5
           Piecewise construction
                                                                                      [pair.piecewise]
   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
                                                                                                 [tuple]
          Tuples
   20.4.1 In general
                                                                                       [tuple.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
```

§ 20.4.1 480

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.

2 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<VTypes...> make_tuple(Types&&...);
  template <class... Types>
    tuple<Types...> forward_as_tuple(Types&&...) noexcept;
 template<class... Types>
    tuple<Types&...> tie(Types&...) noexcept;
 template <class... Tuples>
    tuple<Ctypes...> tuple_cat(Tuples&&...);
  // 20.4.2.5, tuple helper classes:
  template <class T> class tuple_size; // undefined
  template <class T> class tuple_size<const T>;
  template <class T> class tuple_size<volatile T>;
  template <class T> class tuple_size<const volatile T>;
  template <class... Types> class tuple_size<tuple<Types...> >;
  template <size_t I, class T> class tuple_element;
  template <size_t I, class T> class tuple_element<I, const T>;
  template <size_t I, class T> class tuple_element<I, volatile T>;
  template <size_t I, class T> class tuple_element<I, const volatile T>;
  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...>&) noexcept;
  template <size_t I, class... types>
    typename tuple_element<I, tuple<Types...> >::type&& get(tuple<Types...>&&) noexcept;
  template <size_t I, class... types>
    typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>&) noexcept;
  // 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...>&);
  template<class... TTypes, class... UTypes>
    bool operator!=(const tuple<TTypes...>&, const tuple<UTypes...>&);
  template<class... TTypes, class... UTypes>
    bool operator>(const tuple<TTypes...>&, const tuple<UTypes...>&);
 template<class... TTypes, class... UTypes>
    bool operator<=(const tuple<TTypes...>&, const tuple<UTypes...>&);
```

§ 20.4.1 481

```
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) noexcept(see below);
20.4.2 Class template tuple
                                                                                       [tuple.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&&) = default;
      template <class... UTypes>
        tuple(const tuple<UTypes...>&);
      template <class... UTypes>
        tuple(tuple<UTypes...>&&);
      template <class U1, class U2>
                                           // iff sizeof...(Types) == 2
        tuple(const pair<U1, U2>&);
      template <class U1, class U2>
                                           //iff sizeof...(Types) == 2
        tuple(pair<U1, U2>&&);
      // allocator-extended constructors
      template <class Alloc>
        tuple(allocator_arg_t, const Alloc& a);
      template <class Alloc>
        tuple(allocator_arg_t, const Alloc& a, const Types&...);
      template <class Alloc, class... UTypes>
        tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
      template <class Alloc>
        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>
```

§ 20.4.2 482

tuple(allocator\_arg\_t, const Alloc& a, pair<U1, U2>&&);

11

```
// 20.4.2.2, tuple assignment
          tuple& operator=(const tuple&);
          tuple& operator=(tuple&&) noexcept(see below);
          template <class... UTypes>
            tuple& operator=(const tuple<UTypes...>&);
          template <class... UTypes>
            tuple& operator=(tuple<UTypes...>&&);
          template <class U1, class U2>
                                                         // iff sizeof...(Types) == 2
            tuple& operator=(const pair<U1, U2>&);
          template <class U1, class U2>
                                                         // iff sizeof...(Types) == 2
            tuple& operator=(pair<U1, U2>&&);
          // 20.4.2.3, tuple swap
          void swap(tuple&) noexcept(see below);
      }
              Construction
    20.4.2.1
                                                                                                [tuple.cnstr]
<sup>1</sup> For each tuple constructor, an exception is thrown only if the construction of one of the types in Types
    throws an exception.
   In the constructor descriptions that follow, let i be in the range [0,sizeof...(Types)) in order, T_i be the
    i^{th} type in Types, and U_i be the i^{th} type in a template parameter pack named UTypes, where indexing is
    zero-based.
    constexpr tuple();
3
          Requires: is_default_constructibleT_i::value is true for all i.
4
         Effects: Value initializes each element.
    explicit tuple(const Types&...);
5
         Requires: is_copy_constructible\langle T_i \rangle::value is true for all i.
6
          Effects: Initializes each element with the value of the corresponding parameter.
    template <class... UTypes>
      explicit tuple(UTypes&&... u);
7
         Requires: sizeof...(Types) == sizeof...(UTypes). is_constructible < T_i, U_i & ::value is true
8
         Effects: Initializes the elements in the tuple with the corresponding value in std::forward<UTypes>(u).
9
         Remark: This constructor shall not participate in overload resolution unless each type in UTypes is
         implicitly convertible to its corresponding type in Types.
    tuple(const tuple& u) = default;
10
          Requires: is_copy_constructibleT_i::value is true for all i.
         Effects: Initializes each element of *this with the corresponding element of u.
    § 20.4.2.1
                                                                                                           483
```

```
tuple(tuple&& u) = default;
12
          Requires: is_move_constructible\langle T_i \rangle::value is true for all i.
13
         Effects: For all i, initializes the i^{th} element of *this with std::forward<T_i>(get<i>(u)).
    template <class... UTypes> tuple(const tuple<UTypes...>& u);
14
          Requires: sizeof...(Types) == sizeof...(UTypes). is_constructible< T_i, const U_i \& > :: value
         is true for all i.
15
         Effects: Constructs each element of *this with the corresponding element of u.
16
         Remark: This constructor shall not participate in overload resolution unless const U_i& is implicitly
         convertible to T_i for all i.
    template <class... UTypes> tuple(tuple<UTypes...>&& u);
17
          Requires: sizeof...(Types) == sizeof...(UTypes). is_constructibleT_i, U_i&&>::value is true
         Effects: For all i, initializes the i^{th} element of *this with std::forward\langle U_i \rangle(get\langle i \rangle(u)).
18
19
          Remark: This constructor shall not participate in overload resolution unless each type in UTypes is
         implicitly convertible to its corresponding type in Types.
    template <class U1, class U2> tuple(const pair<U1, U2>& u);
20
         Requires: sizeof...(Types) == 2. is_constructible< T_0, const U1&>::value is true for the first
         type T_0 in Types and is_constructible<T_1, const U2&>::value is true for the second type T_1 in
         Types.
21
         Effects: Constructs the first element with u.first and the second element with u.second.
22
         Remark: This constructor shall not participate in overload resolution unless const U1& is implicitly
         convertible to T_0 and const U2& is implicitly convertible to T_1.
    template <class U1, class U2> tuple(pair<U1, U2>&& u);
23
          Requires: sizeof...(Types) == 2. is_constructible< T_0, U1&&>::value is true for the first type
         T_0 in Types and is_constructible<T_1, U2&&>::value is true for the second type T_1 in Types.
24
          Effects: Initializes the first element with std::forward<U1>(u.first) and the second element with
         std::forward<U2>(u.second).
25
          Remark: This constructor shall not participate in overload resolution unless U1 is implicitly convertible
         to T_0 and U2 is implicitly convertible to T_1.
    template <class Alloc>
      tuple(allocator_arg_t, const Alloc& a);
    template <class Alloc>
      tuple(allocator_arg_t, const Alloc& a, const Types&...);
    template <class Alloc, class... UTypes>
      tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
    template <class Alloc>
      tuple(allocator_arg_t, const Alloc& a, const tuple&);
    § 20.4.2.1
                                                                                                           484
```

```
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>&&);
26
         Requires: Alloc shall meet the requirements for an Allocator (17.6.3.5).
27
         Effects: Equivalent to the preceding constructors except that each element is constructed with uses-
         allocator construction (20.6.7.2).
    20.4.2.2
             Assignment
                                                                                               [tuple.assign]
   For each tuple assignment operator, an exception is thrown only if the assignment of one of the types in
    Types throws an exception. In the function descriptions that follow, let i be in the range [0, sizeof...
    (Types)) in order, T_i be the i^{th} type in Types, and U_i be the i^{th} type in a template parameter pack named
    UTypes, where indexing is zero-based.
    tuple& operator=(const tuple& u);
2
         Requires: is_copy_assignableT_i::value is true for all i.
3
         Effects: Assigns each element of u to the corresponding element of *this.
 4
         Returns: *this
    tuple& operator=(tuple&& u) noexcept(see below);
5
         Remark: The expression inside noexcept is equivalent to the logical AND of the following expressions:
           \verb|is_nothrow_move_assignable<|T_i>:: \verb|value||
         where T_i is the i^{th} type in Types.
6
         Requires: is_move_assignableT_i::value is true for all i.
7
         Effects: For all i, assigns std::forwardT_i>(geti>(u)) to geti>(*this).
8
         Returns: *this.
    template <class... UTypes>
      tuple& operator=(const tuple<UTypes...>& u);
9
         Requires: sizeof...(Types) == sizeof...(UTypes) and is_assignable\langle T_i \&, const U_i \& >::value
         is true for all i.
10
         Effects: Assigns each element of u to the corresponding element of *this.
11
         Returns: *this
    template <class... UTypes>
      tuple& operator=(tuple<UTypes...>&& u);
```

§ 20.4.2.2 485

```
12
          Requires: is_assignable<Ti&, Ui&&>::value == true for all i. sizeof...(Types) ==
          sizeof...(UTypes).
13
          Effects: For all i, assigns std::forward\langle U_i \rangle(get\langle i \rangle) to get\langle i \rangle(*this).
14
          Returns: *this.
    template <class U1, class U2> tuple& operator=(const pair<U1, U2>& u);
15
          Requires: sizeof...(Types) == 2. is assignable< T_0 \&, const U1&>::value is true for the first
         type T_0 in Types and is_assignable<T_1&, const U2&>::value is true for the second type T_1 in
         Types.
16
          Effects: Assigns u.first to the first element of *this and u.second to the second element of *this.
17
          Returns: *this
    template <class U1, class U2> tuple& operator=(pair<U1, U2>&& u);
18
          Requires: sizeof...(Types) == 2. is_assignable< T_0 \&, U1&&>::value is true for the first type T_0
         in Types and is_assignable< T_1 \&, U2&&>::value is true for the second type T_1 in Types.
19
          Effects: Assigns std::forward<U1>(u.first) to the first element of *this and
         std::forward<U2>(u.second) to the second element of *this.
20
          Returns: *this.
    20.4.2.3
               swap
                                                                                                 [tuple.swap]
    void swap(tuple& rhs) noexcept(see below);
 1
          Remark: The expression inside noexcept is equivalent to the logical AND of the following expressions:
            noexcept(swap(declval< T_i \&>>(), declval< T_i \&>()))
         where T_1 is the i^{th} type in Types.
 2
          Requires: Each element in *this shall be swappable with (17.6.3.2) the corresponding element in rhs.
 3
          Effects: Calls swap for each element in *this and its corresponding element in rhs.
 4
          Throws: Nothing unless one of the element-wise swap calls throws an exception.
    20.4.2.4 Tuple creation functions
                                                                                             [tuple.creation]
   In the function descriptions that follow, let i be in the range [0,sizeof...(TTypes)) in order and let T_i
    be the i^{th} type in a template parameter pack named TTypes; let j be in the range [0,sizeof...(UTypes))
    in order and U_i be the j^{th} type in a template parameter pack named UTypes, where indexing is zero-based.
    template<class... Types>
      tuple<VTypes...> make_tuple(Types&&... t);
 2
         Let U_i be decayT_i > :: type for each T_i in Types. Then each V_i in VTypes is X& if U_i equals reference_-
         wrapper<X>, otherwise V_i is U_i.
 3
          Returns: tuple<VTypes...>(std::forward<Types>(t)...).
 4
         [Example:
            int i; float j;
            make_tuple(1, ref(i), cref(j))
         creates a tuple of type
    § 20.4.2.4
                                                                                                           486
```

Effects: Constructs a tuple of references to the arguments in t suitable for forwarding as arguments to a function. Because the result may contain references to temporary variables, a program shall ensure that the return value of this function does not outlive any of its arguments. (e.g., the program should typically not store the result in a named variable).

Returns: tuple<Types&&...>(std::forward<Types>(t)...)

```
template<class... Types>
  tuple<Types&...> tie(Types&... t) noexcept;
```

6

- Returns: tuple<Types&>(t...). When an argument in t is 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:

```
int i; std::string s;
  tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
  // i == 42, s == "C++"
  -- end example]

template <class... Tuples>
  tuple<CTypes...> tuple_cat(Tuples&&... tpls);
```

- In the following paragraphs, let  $T_i$  be the  $i^{th}$  type in Tuples,  $U_i$  be remove\_reference<Ti>::type, and  $tp_i$  be the  $i^{th}$  parameter in the function parameter pack tpls, where all indexing is zero-based.
- Requires: For all i,  $U_i$  shall be the type  $cv_i$  tuple $\langle Args_i...\rangle$ , where  $cv_i$  is the (possibly empty)  $i^{th}$  cv-qualifier-seq and  $Args_i$  is the parameter pack representing the element types in  $U_i$ . Let  $A_{ik}$  be the  $k_i^{th}$  type in  $Args_i$ . For all  $A_{ik}$  the following requirements shall be satisfied: If  $T_i$  is deduced as an lvalue reference type, then is\_constructible $\langle A_{ik}, cv_i A_{ik} \rangle$ ::value == true, otherwise is\_constructible $\langle A_{ik}, cv_i A_{ik} \rangle$ ::value == true.
- Remarks: The types in Ctypes shall be equal to the ordered sequence of the extended types  $Args_0...$ ,  $Args_{n-1}...$ , where n is equal to sizeof...(Tuples). Let  $e_i...$  be the  $i^{th}$  ordered sequence of tuple elements of the resulting tuple object corresponding to the type sequence  $Args_i$ .
- Returns: A tuple object constructed by initializing the  $k_i^{th}$  type element  $e_{ik}$  in  $e_i$ ... with get $< k_i > (std::forward < T_i > (tp_i))$  for each valid  $k_i$  and each group  $e_i$  in order.
- Note: An implementation may support additional types in the parameter pack Tuples that support the tuple-like protocol, such as pair and array.

§ 20.4.2.4 487

[tuple.helper]

20.4.2.5

1

Tuple helper classes

```
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: I < sizeof...(Types). The program is ill-formed if I is out of bounds.
        Type: TI is the type of the Ith element of Types, where indexing is zero-based.
   template <class T> class tuple_size<const T>;
   template <class T> class tuple_size<volatile T>;
   template <class T> class tuple_size<const volatile T>;
        Let TS denote tuple_size<T> of the cv-unqualified type T. Then each of the three templates shall
        meet the UnaryTypeTrait requirements (20.9.1) with a BaseCharacteristic of
          integral_constant<remove_cv<decltype(TS::value)>::type, TS::value>
   template <size_t I, class T> class tuple_element<I, const T>;
   template <size_t I, class T> class tuple_element<I, volatile T>;
   template <size_t I, class T> class tuple_element<I, const volatile T>;
        Let TE denote tuple element I, T of the cv-unqualified type T. Then each of the three templates
        shall meet the TransformationTrait requirements (20.9.1) with a member typedef type that names
        the following type:
         — for the first specialization, add_const<TE::type>::type,
         — for the second specialization, add_volatile<TE::type>::type, and
          — for the third specialization, add_cv<TE::type>::type.
             Element access
   20.4.2.6
                                                                                            [tuple.elem]
   template <size_t I, class... Types>
     typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>& t) noexcept;
        Requires: I < sizeof...(Types). The program is ill-formed if I is out of bounds.
        Returns: A reference to the Ith element of t, where indexing is zero-based.
   template <size_t I, class... types>
     typename tuple_element<I, tuple<Types...> >::type&& get(tuple<Types...>&& t) noexcept;
3
        Effects: Equivalent to return std::forward<typename tuple_element<I, tuple<Types...> >
        ::type&&>(get<I>(t));
4
        Note: if a T in Types is some reference type X&, the return type is X&, not X&&. However, if the element
        type is a non-reference type T, the return type is T&&.
```

§ 20.4.2.6 488

5

6

7

8

1

3

4

5

7

```
template <size_t I, class... Types>
  typename tuple element<I, tuple<Types...> >::type const& get(const tuple<Types...>& t) noexcept;
     Requires: I < sizeof...(Types). The program is ill-formed if I is out of bounds.
     Returns: A const reference to the 1th element of t, where indexing is zero-based.
     [Note: Constness is shallow. If a T in Types is some reference type X&, the return type is X&, not
     const X&. However, if the element type is non-reference type T, the return type is const T&. This is
     consistent with how constness is defined to work for member variables of reference type. — end note]
     Note: The reason get is a nonmember function is that if this functionality had been provided as a
     member function, code where the type depended on a template parameter would have required using
     the template keyword. — end note]
20.4.2.7 Relational operators
                                                                                           [tuple.rel]
template<class... TTypes, class... UTypes>
 bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
     Requires: For all i, where 0 <= i and i < sizeof...(Types), get<i>(t) == get<i>(u) is a valid
     expression returning a type that is convertible to bool. sizeof...(TTypes) == sizeof...(UTypes).
     Returns: true iff get<i>(t) == get<i>(u) for all i. For any two zero-length tuples e and f, e == f
     returns true.
     Effects: The elementary comparisons are performed in order from the zeroth index upwards. No
     comparisons or element accesses are performed after the first equality comparison that evaluates to
     false.
template < class... TTypes, class... UTypes>
  bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
     Requires: For all i, where 0 <= i and i < sizeof...(Types), get<i>(t) < get<i>(u) and get<i>(u)
     < get<i>(t) are valid expressions returning types that are convertible to bool. sizeof...(TTypes)
     == sizeof...(UTypes).
     Returns: The result of a lexicographical comparison between t and u. The result is defined as:
     (bool)(get<0>(t) < get<0>(u) | (!(bool)(get<0>(u) < get<0>(t)) && ttail < utall, where
     r<sub>tail</sub> for some tuple r is a tuple containing all but the first element of r. For any two zero-length tuples
     e and f, e < f returns false.
template<class... TTypes, class... UTypes>
 bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
     Returns: !(t == u).
template<class... TTypes, class... UTypes>
 bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
     Returns: u < t.
template<class... TTypes, class... UTypes>
 bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
     Returns: !(u < t)
§ 20.4.2.7
                                                                                                   489
```

```
template<class... TTypes, class... UTypes>
     bool operator>=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
        Returns: !(t < u)
   Note: The above definitions for comparison operators do not require t<sub>tail</sub> (or u<sub>tail</sub>) to be constructed. It may
   not even be possible, as t and u are not required to be copy constructible. Also, all comparison operators
   are short circuited; they do not perform element accesses beyond what is required to determine the result
   of the comparison. -end note
   20.4.2.8
             Tuple traits
                                                                                             [tuple.traits]
   template <class... Types, class Alloc>
     struct uses_allocator<tuple<Types...>, Alloc> : true_type { };
        Requires: Alloc shall be an Allocator (17.6.3.5).
        Note: Specialization of this trait informs other library components that tuple can be constructed
        with an allocator, even though it does not have a nested allocator_type. — end note]
   20.4.2.9
              Tuple specialized algorithms
                                                                                            [tuple.special]
   template <class... Types>
     void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);
        Remark: The expression inside noexcept is equivalent to:
          noexcept(x.swap(y))
2
        Effects: x.swap(y)
         Class template bitset
                                                                                       [template.bitset]
      Header <br/>
<br/>
bitset> synopsis
     #include <string>
                                      // for istream, ostream
     #include <iosfwd>
     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>&) noexcept;
       template <size_t N>
         bitset<N> operator | (const bitset<N>&, const bitset<N>&) noexcept;
       template <size_t N>
         bitset<N> operator^(const bitset<N>&, const bitset<N>&) noexcept;
       template <class charT, class traits, size_t N>
         basic_istream<charT, traits>&
         operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
       template <class charT, class traits, size_t N>
         basic_ostream<charT, traits>&
         operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
```

§ 20.5

The header <br/> defines a class template and several related functions for representing and manipulating

fixed-size sequences of bits.

```
namespace std {
  template<size_t N> class bitset {
  public:
    // bit reference:
    class reference {
      friend class bitset;
      reference() noexcept;
    public:
     ~reference() noexcept;
      reference& operator=(bool x) noexcept;
                                                           // for b[i] = x;
                                                           // for b[i] = b[j];
      reference& operator=(const reference&) noexcept;
                                                           // flips the bit
      bool operator~() const noexcept;
                                                           // for x = b[i];
      operator bool() const noexcept;
      reference& flip() noexcept;
                                                           // for b[i].flip();
    };
    // 20.5.1 constructors:
    constexpr bitset() noexcept;
    constexpr bitset(unsigned long long val) noexcept;
    template < class charT, class traits, class Allocator >
      explicit bitset(
        const basic_string<charT,traits,Allocator>& str,
        typename basic_string<charT,traits,Allocator>::size_type pos = 0,
        typename basic_string<charT,traits,Allocator>::size_type n =
          basic_string<charT, traits, Allocator>::npos,
          charT zero = charT('0'), charT one = charT('1'));
    template <class charT>
      explicit bitset(
        const charT* str,
        typename basic_string<charT>::size_type n = basic_string<charT>::npos,
        charT zero = charT('0'), charT one = charT('1'));
    // 20.5.2 bitset operations:
    bitset<N>& operator&=(const bitset<N>& rhs) noexcept;
    bitset<N>& operator|=(const bitset<N>& rhs) noexcept;
    bitset<N>& operator^=(const bitset<N>& rhs) noexcept;
    bitset<N>& operator<<=(size_t pos) noexcept;</pre>
    bitset<N>& operator>>=(size_t pos) noexcept;
    bitset<N>& set() noexcept;
    bitset<N>& set(size_t pos, bool val = true);
    bitset<N>& reset() noexcept;
    bitset<N>& reset(size_t pos);
    bitset<N> operator~() const noexcept;
    bitset<N>& flip() noexcept;
    bitset<N>& flip(size_t pos);
    // element access:
                                                        // for b[i];
    constexpr bool operator[](size_t pos) const;
    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> >
```

§ 20.5

```
basic_string<charT, traits, Allocator>
    to_string(charT zero = charT('0'), charT one = charT('1')) const;
  size_t count() const noexcept;
  constexpr size_t size() noexcept;
 bool operator==(const bitset<N>& rhs) const noexcept;
 bool operator!=(const bitset<N>& rhs) const noexcept;
 bool test(size_t pos) const;
 bool all() const noexcept;
 bool any() const noexcept;
 bool none() const noexcept;
 bitset<N> operator<<(size_t pos) const noexcept;</pre>
 bitset<N> operator>>(size_t pos) const noexcept;
};
// 20.5.3 hash support
template <class T> struct hash;
template <size_t N> struct hash<bitset<N> >;
```

- The class template bitset<N>describes an object that can store a sequence consisting of a fixed number of bits, N.
- <sup>3</sup> 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.
- <sup>4</sup> 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

[bitset.cons]

constexpr bitset() noexcept;

1

2

Effects: Constructs an object of class bitset<N>, initializing all bits to zero.

constexpr bitset(unsigned long long val) noexcept;

Effects: Constructs an object of class bitset<N>, initializing the first M bit positions to the corresponding bit values in val. M is the smaller of N and the number of bits in the value representation (3.9) of unsigned long long. If M < N, the remaining bit positions are initialized to zero.

§ 20.5.1 492

```
3 Requires: pos <= str.size().</pre>
```

- 4 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 str, beginning at position pos, is 0 zero. Otherwise, the element has the value 1. Character position pos + M 1 corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions.
- <sup>7</sup> If M < N, remaining bit positions are initialized to zero.

```
template <class charT>
    explicit bitset(
    const charT* str,
    typename basic_string<charT>::size_type n = basic_string<charT>::npos,
    charT zero = charT('0'), charT one = charT('1'));

**Effects: Constructs an object of class bitset<N> as if by

bitset(
    n == basic_string<charT>::npos
    ? basic_string<charT>(str)
    : basic_string<charT>(str, n),
    0, n, zero, one)
```

# 20.5.2 bitset members

[bitset.members]

bitset<N>& operator&=(const bitset<N>& rhs) noexcept;

Effects: Clears each bit in \*this for which the corresponding bit in rhs is clear, and leaves all other bits unchanged.

2 Returns: \*this.

1

bitset<N>& operator|=(const bitset<N>& rhs) noexcept;

- Effects: Sets each bit in \*this for which the corresponding bit in rhs is set, and leaves all other bits unchanged.
- 4 Returns: \*this.

bitset<N>& operator^=(const bitset<N>& rhs) noexcept;

- Effects: Toggles each bit in \*this for which the corresponding bit in rhs is set, and leaves all other bits unchanged.
- 6 Returns: \*this.

§ 20.5.2 493

```
bitset<N>& operator<<=(size_t pos) noexcept;</pre>
7
          Effects: Replaces each bit at position I in *this with a value determined as follows:
           — If I < pos, the new value is zero;
           — If I >= pos, the new value is the previous value of the bit at position I - pos.
          Returns: *this.
    bitset<N>& operator>>=(size_t pos) noexcept;
9
         Effects: Replaces each bit at position I in *this with a value determined as follows:
           — If pos >= N - I, the new value is zero;
           — If pos < N - I, the new value is the previous value of the bit at position I + pos.
10
          Returns: *this.
    bitset<N>& set() noexcept;
11
          Effects: Sets all bits in *this.
12
          Returns: *this.
    bitset<N>& set(size_t pos, bool val = true);
13
          Requires: pos is valid
14
          Throws: out_of_range if pos does not correspond to a valid bit position.
          Effects: Stores a new value in the bit at position pos in *this. If val is nonzero, the stored value is
         one, otherwise it is zero.
16
          Returns: *this.
    bitset<N>& reset() noexcept;
17
          Effects: Resets all bits in *this.
18
          Returns: *this.
    bitset<N>& reset(size_t pos);
19
          Requires: pos is valid
20
          Throws: out_of_range if pos does not correspond to a valid bit position.
21
          Effects: Resets the bit at position pos in *this.
22
          Returns: *this.
    bitset<N> operator~() const noexcept;
23
          Effects: Constructs an object x of class bitset<N> and initializes it with *this.
         Returns: x.flip().
    § 20.5.2
                                                                                                            494
```

```
bitset<N>& flip() noexcept;
^{25}
         Effects: Toggles all bits in *this.
26
          Returns: *this.
    bitset<N>& flip(size_t pos);
27
         Requires: pos is valid
28
          Throws: out_of_range if pos does not correspond to a valid bit position.
29
         Effects: Toggles the bit at position pos in *this.
30
          Returns: *this.
    unsigned long to_ulong() const;
31
          Throws: overflow_error if the integral value x corresponding to the bits in *this cannot be repre-
         sented as type unsigned long.
32
          Returns: x.
    unsigned long long to_ullong() const;
33
          Throws: overflow_error if the integral value x corresponding to the bits in *this cannot be repre-
         sented as type unsigned long long.
34
          Returns: x.
    template <class charT = char,</pre>
        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;
35
         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.
36
         Returns: The created object.
    size_t count() const noexcept;
37
         Returns: A count of the number of bits set in *this.
    constexpr size_t size() noexcept;
38
         Returns: N.
    bool operator==(const bitset<N>& rhs) const noexcept;
                                                                                                          495
    § 20.5.2
```

```
39
         Returns: true if the value of each bit in *this equals the value of the corresponding bit in rhs.
    bool operator!=(const bitset<N>& rhs) const noexcept;
40
         Returns: true if !(*this == rhs).
    bool test(size_t pos) const;
41
         Requires: pos is valid
42
         Throws: out_of_range if pos does not correspond to a valid bit position.
43
         Returns: true if the bit at position pos in *this has the value one.
    bool all() const noexcept;
44
         Returns: count() == size()
    bool any() const noexcept;
45
         Returns: count() != 0
    bool none() const noexcept;
46
         Returns: count() == 0
    bitset<N> operator<<(size_t pos) const noexcept;</pre>
47
         Returns: bitset<N>(*this) <<= pos.</pre>
    bitset<N> operator>>(size_t pos) const noexcept;
48
         Returns: bitset<N>(*this) >>= pos.
    constexpr bool operator[](size_t pos);
49
         Requires: pos shall be valid.
50
         Returns: true if the bit at position pos in *this has the value one, otherwise false.
51
         Throws: Nothing.
    bitset<N>::reference operator[](size_t pos);
52
         Requires: pos shall be valid.
         Returns: An object of type bitset<N>::reference such that (*this)[pos] == this->test(pos),
53
         and such that (*this)[pos] = val is equivalent to this->set(pos, val).
54
         Throws: Nothing.
55
         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.
```

496

§ 20.5.2

```
[bitset.hash]
   20.5.3 bitset hash support
   template <size_t N> struct hash<bitset<N> >;
        Requires: the template specialization shall meet the requirements of class template hash (20.8.12).
                                                                                   [bitset.operators]
   20.5.4 bitset operators
   bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
        Returns: bitset<N>(lhs) &= rhs.
   bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
2
        Returns: bitset<N>(lhs) |= rhs.
   \verb|bitset<N> a lhs, const bitset<N> & rhs) noexcept; \\
3
        Returns: bitset<N>(lhs) ^= rhs.
   template <class charT, class traits, size_t N>
     basic_istream<charT, traits>&
     operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
4
        A formatted input function (27.7.2.2).
5
        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).
6
        If no characters are stored in str, calls is.setstate(ios base::failbit) (which may throw ios -
        base::failure (27.5.5.4)).
        Returns: is.
   template <class charT, class traits, size_t N>
     basic_ostream<charT, traits>&
     operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
        Returns:
          os << x.template to_string<charT,traits,allocator<charT> >(
            use_facet<ctype<charT> >(os.getloc()).widen('0'),
            use_facet<ctype<charT> >(os.getloc()).widen('1'))
```

§ 20.5.4 497

(see 27.7.3.6).

20.6 Memory [memory]

### 20.6.1 In general

[memory.general]

This subclause describes the contents of the header <memory> (20.6.2) and some of the contents of the C headers <cstdlib> and cstring> (20.6.13).

#### 20.6.2 Header <memory> synopsis

[memory.syn]

The header <memory> defines several types and function templates that describe properties of pointers and pointer-like types, manage memory for containers and other template types, and construct multiple objects in uninitialized memory buffers (20.6.3-20.6.12). The header also defines the templates unique\_ptr, shared\_ptr, weak\_ptr, and various template functions that operate on objects of these types (20.7).

```
namespace std {
  // 20.6.3, pointer traits
 template <class Ptr> struct pointer_traits;
  template <class T> struct pointer_traits<T*>;
  // 20.6.4, pointer safety
  enum class pointer_safety { relaxed, preferred, strict };
  void declare reachable(void *p);
 template <class T> T *undeclare_reachable(T *p);
  void declare_no_pointers(char *p, size_t n);
  void undeclare_no_pointers(char *p, size_t n);
  pointer_safety get_pointer_safety() noexcept;
  // 20.6.5, pointer alignment function
  void *align(std::size_t alignment, std::size_t size,
    void *&ptr, std::size_t& space);
  // 20.6.6, allocator argument tag
  struct allocator_arg_t { };
  constexpr allocator_arg_t allocator_arg = allocator_arg_t();
  // 20.6.7, uses_allocator
  template <class T, class Alloc> struct uses_allocator;
  // 20.6.8, allocator traits
  template <class Alloc> struct allocator_traits;
  // 20.6.9, 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>&) noexcept;
  template <class T, class U>
    bool operator!=(const allocator<T>&, const allocator<U>&) noexcept;
  // 20.6.10, raw storage iterator:
  template <class OutputIterator, class T> class raw_storage_iterator;
  // 20.6.11, temporary buffers:
  template <class T>
    pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n) noexcept;
  template <class T>
    void return_temporary_buffer(T* p);
```

§ 20.6.2 498

```
// 20.6.12, specialized algorithms:
template <class T> T* addressof(T& r) noexcept;
template <class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
template <class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);
template <class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                          const T& x);
template <class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
// 20.7.1 class template unique_ptr:
template <class T> struct default_delete;
template <class T> struct default_delete<T[]>;
template <class T, class D = default_delete<T>> class unique_ptr;
template <class T, class D> class unique_ptr<T[], D>;
template <class 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);
template <class T, class D>
  bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator == (nullptr_t, const unique_ptr<T, D>& y) noexcept;
template <class T, class D>
  bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator!=(nullptr_t, const unique_ptr<T, D>& y) noexcept;
template <class T, class D>
  bool operator<(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<=(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
```

§ 20.6.2

```
template <class T, class D>
  bool operator>=(nullptr_t, const unique_ptr<T, D>& y);
// 20.7.2.1, class bad_weak_ptr:
class bad_weak_ptr;
// 20.7.2.2, class template shared ptr:
template<class T> class shared_ptr;
// 20.7.2.2.7, shared_ptr comparisons:
template < class T, class U>
 bool operator==(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template < class T, class U>
  bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U> \,
  bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template<class T, class U>
  bool operator>(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template < class T, class U>
  bool operator<=(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template < class T, class U>
  bool operator>=(shared_ptr<T> const& a, shared_ptr<U> const& b) noexcept;
template <class T>
  bool operator==(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator==(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator!=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator!=(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator<(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator<(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator<=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator<=(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator>(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator>=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& y) noexcept;
// 20.7.2.2.8, shared_ptr specialized algorithms:
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;
// 20.7.2.2.9, shared_ptr casts:
template<class T, class U>
  shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r) noexcept;
template < class T, class U>
```

§ 20.6.2

```
shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r) noexcept;
template<class T, class U>
  shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r) noexcept;
// 20.7.2.2.10, shared_ptr get_deleter:
template<class D, class T> D* get_deleter(shared_ptr<T> const& p) noexcept;
// 20.7.2.2.11, shared ptr I/O:
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);
// 20.7.2.3, class template weak_ptr:
template<class T> class weak_ptr;
// 20.7.2.3.6, weak_ptr specialized algorithms:
\label{template} $$ $$ $$ template < class T> void $$ swap(weak_ptr < T> \& a, weak_ptr < T> \& b) noexcept;
// 20.7.2.3.7, class template owner_less:
template<class T> class owner_less;
// 20.7.2.4, class template enable_shared_from_this:
template<class T> class enable_shared_from_this;
// 20.7.2.5, 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(
```

§ 20.6.2 501

```
shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
        memory_order success, memory_order failure);
    // 20.7.2.6 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> >;
    // D.10, auto_ptr (deprecated)
    template <class X> class auto_ptr;
20.6.3
        Pointer traits
                                                                                     [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 {
      typedef Ptr
                       pointer;
       typedef see below element_type;
       typedef see below difference_type;
      template <class U> using rebind = see below;
      static pointer pointer_to(see below r);
    };
    template <class T> struct pointer_traits<T*> {
       typedef T*
                        pointer;
       typedef T
                         element_type;
      typedef ptrdiff_t difference_type;
       template <class U> using rebind = U*;
      static pointer_to(see below r) noexcept;
    };
20.6.3.1
         Pointer traits member types
                                                                                [pointer.traits.types]
typedef see below element_type;
      Type: Ptr::element_type if such a type exists; otherwise, T if Ptr is a class template instantia-
      tion 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;
      Alias template: 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.
```

§ 20.6.3.1 502

2

3

#### 20.6.3.2 Pointer traits member functions

[pointer.traits.functions]

```
static pointer_pointer_traits::pointer_to(see below r);
static pointer_pointer_traits<T*>::pointer_to(see below r) noexcept;
```

Remark: if element\_type is (possibly cv-qualified) void, the type of r is unspecified; otherwise, it is element\_type&.

Returns: The first member 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 member function returns std::addressof(r).

## 20.6.4 Pointer safety

[util.dynamic.safety]

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.

void declare\_reachable(void \*p);

- 2 Requires: p shall be a safely-derived pointer (3.7.4.3) or a null pointer value.
- 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.

template <class T> T \*undeclare\_reachable(T \*p);

- 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.
- [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]

```
void declare_no_pointers(char *p, size_t n);
```

- Requires: No bytes in the specified range are currently registered with declare\_no\_pointers(). If the specified range is in an allocated object, then it must be entirely within a single allocated object. The object must be live until the corresponding undeclare\_no\_pointers() call. [Note: In a garbage-collecting implementation, the fact that a region in an object is registered with declare\_no\_pointers() should not prevent the object from being collected. end note]
- 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 by global operator new and not previously declared reachable. [Note: This may be used to inform a garbage collector or leak detector that this region of memory need not be traced. —end note]
- 11 Throws: Nothing.
- [Note: Under some conditions implementations may need to allocate memory. However, the request can be ignored if memory allocation fails. -end note]

§ 20.6.4 503

```
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.
- 15 Throws: Nothing.

```
pointer_safety get_pointer_safety() noexcept;
```

Returns: pointer\_safety::strict if the implementation has strict pointer safety (3.7.4.3). It is implementation defined whether get\_pointer\_safety returns pointer\_safety::relaxed or pointer\_safety::preferred if the implementation has relaxed pointer safety.<sup>229</sup>

20.6.5 Align [ptr.align]

```
void *align(std::size_t alignment, std::size_t size,
   void *&ptr, std::size_t& space);
```

Effects: If it is possible to fit size bytes of storage aligned by alignment into the buffer pointed to by ptr with length space, the function updates ptr to point to the first possible address of such storage and decreases space by the number of bytes used for alignment. Otherwise, the function does nothing.

- 2 Requires:
  - alignment shall be a fundamental alignment value or an extended alignment value supported by the implementation in this context
  - ptr shall point to contiguous storage of at least space bytes
- Returns: A null pointer if the requested aligned buffer would not fit into the available space, otherwise the adjusted value of ptr.
- [Note: The function updates its ptr and space arguments so that it can be called repeatedly with possibly different alignment and size arguments for the same buffer.

### 20.6.6 Allocator argument tag

[allocator.tag]

```
namespace std {
   struct allocator_arg_t { };
   constexpr allocator_arg_t allocator_arg = allocator_arg_t();
}
```

The allocator\_arg\_t struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see tuple 20.4) have constructors with allocator\_arg\_t as the first argument, immediately followed by an argument of a type that satisfies the Allocator requirements (17.6.3.5).

```
20.6.7 uses_allocator
```

[allocator.uses]

20.6.7.1 uses\_allocator trait

[allocator.uses.trait]

template <class T, class Alloc> struct uses\_allocator;

§ 20.6.7.1 504

<sup>229)</sup> pointer\_safety::preferred might be returned to indicate that a leak detector is running so that the program can avoid spurious leak reports.

Remark: automatically detects whether T has a nested allocator\_type that is convertible from Alloc. Meets the BinaryTypeTrait requirements (20.9.1). 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.

#### 20.6.7.2 uses-allocator construction

## [allocator.uses.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, Alloc>::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]

### 20.6.8 Allocator traits

[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;
```

§ 20.6.8 505

```
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);
      };
    }
   20.6.8.1 Allocator traits member types
                                                                               [allocator.traits.types]
   typedef see below pointer;
        Type: Alloc::pointer if such a type exists; otherwise, value_type*.
   typedef see below const_pointer;
        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>.
   typedef see below difference_type;
5
        Type: Alloc::difference_type if such a type exists; otherwise, pointer_traits<pointer>::dif-
        ference_type.
   typedef see below size_type;
        Type: Alloc::size_type if such a type exists; otherwise, make_unsigned<difference_type>::type.
   typedef see below propagate_on_container_copy_assignment;
```

§ 20.6.8.1 506

```
7
         Type: Alloc::propagate_on_container_copy_assignment if such a type exists, otherwise false_-
         type.
   typedef see below propagate_on_container_move_assignment;
         Type: Alloc::propagate_on_container_move_assignment if such a type exists, otherwise false_-
8
         type.
   typedef see below propagate_on_container_swap;
         Type: Alloc::propagate_on_container_swap if such a type exists, otherwise false_type.
   template <class T> using rebind_alloc = see below;
10
         Alias template: 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 argu-
         ments; otherwise, the instantiation of rebind alloc is ill-formed.
   20.6.8.2
             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).
4
         Throws: Nothing.
   template <class T, class... Args>
      static void construct(Alloc& a, T* p, Args&&... args);
5
         Effects: calls a.construct(p, std::forward<Args>(args)...) if that call is well-formed; otherwise,
         invokes ::new (static_cast<void*>(p)) T(std::forward<Args>(args)...).
   template <class T>
      static void destroy(Alloc& a, T* p);
         Effects: calls a.destroy(p) if that call is well-formed; otherwise, invokes p->~T().
   static size_type max_size(Alloc& a);
7
         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);
8
         Returns: rhs.select_on_container_copy_construction() if that expression is well-formed; other-
         wise, rhs.
```

§ 20.6.8.2 507

#### 20.6.9 The default allocator

[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 {
                      size_type;
   typedef size_t
    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() noexcept;
    allocator(const allocator&) noexcept;
    template <class U> allocator(const allocator<U>&) noexcept;
   ~allocator();
   pointer address(reference x) const noexcept;
    const_pointer address(const_reference x) const noexcept;
   pointer allocate(
      size_type, allocator<void>::const_pointer hint = 0);
    void deallocate(pointer p, size_type n);
    size_type max_size() const noexcept;
    template < class U, class... Args>
      void construct(U* p, Args&&... args);
    template <class U>
      void destroy(U* p);
 };
```

### 20.6.9.1 allocator members

[allocator.members]

Except for the destructor, member functions of the default allocator shall not introduce data races (1.10) as a result of concurrent calls to those member functions from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

pointer address(reference x) const noexcept;

Returns: The actual address of the object referenced by x, even in the presence of an overloaded operator&.

§ 20.6.9.1 508

```
const_pointer address(const_reference x) const noexcept;
3
         Returns: The actual address of the object referenced by x, even in the presence of an overloaded
         operator&.
    pointer allocate(size_type n, allocator<void>::const_pointer hint = 0);
4
         [Note: In a container member function, the address of an adjacent element is often a good choice to
         pass for the hint argument. — end note]
5
         Returns: A pointer to the initial element of an array of storage of size n * sizeof(T), aligned appropri-
         ately for objects of type T. It is implementation-defined whether over-aligned types are supported (3.11).
6
         Remark: the storage is obtained by calling ::operator new(std::size t) (18.6.1), but it is unspec-
         ified when or how often this function is called. The use of hint is unspecified, but intended as an aid
         to locality if an implementation so desires.
7
         Throws: bad alloc if the storage cannot be obtained.
    void deallocate(pointer p, size_type n);
8
         Requires: p shall be a pointer value obtained from allocate(). n shall equal the value passed as the
         first argument to the invocation of allocate which returned p.
9
         Effects: Deallocates the storage referenced by p.
10
         Remarks: Uses::operator delete(void*) (18.6.1), but it is unspecified when this function is called.
    size_type max_size() const noexcept;
11
         Returns: The largest value N for which the call allocate(N,0) might succeed.
    template <class U, class... Args>
      void construct(U* p, Args&&... args);
12
         Effects: ::new((void *)p) U(std::forward<Args>(args)...)
    template <class U>
      void destroy(U* p);
13
         Effects: p->~U()
    20.6.9.2 allocator globals
                                                                                         [allocator.globals]
    template <class T1, class T2>
      bool operator==(const allocator<T1>&, const allocator<T2>&) noexcept;
         Returns: true.
    template <class T1, class T2>
      bool operator!=(const allocator<T1>&, const allocator<T2>&) noexcept;
2
         Returns: false.
```

509

§ 20.6.9.2

#### 20.6.10 Raw storage iterator

template <class OutputIterator, class T>

namespace std {

# [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).

```
class raw_storage_iterator
         : public iterator<output_iterator_tag,void,void,void,void> {
       public:
         explicit raw_storage_iterator(OutputIterator x);
         raw_storage_iterator<OutputIterator,T>& operator*();
         raw_storage_iterator<OutputIterator,T>& operator=(const T& element);
         raw_storage_iterator<OutputIterator,T>& operator++();
         raw_storage_iterator<OutputIterator,T> operator++(int);
       };
     }
   explicit 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*();
3
        Returns: *this
   raw_storage_iterator<OutputIterator,T>& operator=(const T& element);
4
         Effects: Constructs a value from element at the location to which the iterator points.
5
         Returns: A reference to the iterator.
   raw_storage_iterator<OutputIterator,T>& operator++();
6
        Effects: Pre-increment: advances the iterator and returns a reference to the updated iterator.
   raw_storage_iterator<OutputIterator,T> operator++(int);
7
         Effects: Post-increment: advances the iterator and returns the old value of the iterator.
                                                                                     [temporary.buffer]
   20.6.11
              Temporary buffers
   template <class T>
     pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n) noexcept;
1
         Effects: Obtains a pointer to storage sufficient to store up to n adjacent T objects. It is implementation-
        defined whether over-aligned types are supported (3.11).
2
        Returns: A pair containing the buffer's address and capacity (in the units of sizeof(T)), or a pair
        of 0 values if no storage can be obtained or if n \le 0.
   template <class T> void return_temporary_buffer(T* p);
3
         Effects: Deallocates the buffer to which p points.
        Requires: The buffer shall have been previously allocated by get_temporary_buffer.
```

§ 20.6.11 510

# 20.6.12 Specialized algorithms

# [specialized.algorithms]

All the iterators that are used as formal template parameters in the following algorithms are required to have their operator\* return an object for which operator\* is defined and returns a pointer to T. In the algorithm uninitialized\_copy, the formal template parameter InputIterator is required to satisfy the requirements of an input iterator (24.2.3). 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.

```
20.6.12.1 addressof
```

1

[specialized.addressof]

template <class T> T\* addressof(T& r) noexcept;

Returns: The actual address of the object or function referenced by r, even in the presence of an overloaded operator&.

```
20.6.12.2 uninitialized_copy
```

[uninitialized.copy]

```
template <class InputIterator, class ForwardIterator>
     ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                        ForwardIterator result);
1
        Effects:
          for (; first != last; ++result, ++first)
            ::new (static_cast<void*>(&*result))
              typename iterator_traits<ForwardIterator>::value_type(*first);
2
        Returns: result
   template <class InputIterator, class Size, class ForwardIterator>
     ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                           ForwardIterator result);
3
        Effects:
          for (; n > 0; ++result, ++first, --n) {
            ::new (static_cast<void*>(&*result))
              typename iterator_traits<ForwardIterator>::value_type(*first);
          }
        Returns: result
                                                                                      [uninitialized.fill]
   20.6.12.3 uninitialized_fill
   template <class ForwardIterator, class T>
     void uninitialized fill(ForwardIterator first, ForwardIterator last,
                             const T& x);
1
        Effects:
          for (; first != last; ++first)
            ::new (static_cast<void*>(&*first))
              typename iterator_traits<ForwardIterator>::value_type(x);
```

§ 20.6.12.3 511

#### 20.6.12.4 uninitialized\_fill\_n

[uninitialized.fill.n]

# 20.6.13 C library

1

[c.malloc]

Table 45 describes the header <cstdlib>.

Table 45 — Header <cstdlib> synopsis

Type	$\mathbf{Name}(\mathbf{s})$	
Functions:	calloc	malloc
	free	realloc

- <sup>2</sup> 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 46 describes the header <cstring>.

Table 46 — Header <cstring> synopsis

Type	Name(s)	
Macro:	NULL	
Type:	size_t	
Functions:	memchr	memcmp
memcpy	memmove	memset

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.

§ 20.6.13 512

# 20.7 Smart pointers

[smartptr]

# 20.7.1 Class template unique\_ptr

[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).
- <sup>3</sup> 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.
- <sup>4</sup> Additionally, u can, upon request,  $transfer\ ownership$  to another unique pointer u2. Upon completion of such a transfer, the following postconditions hold:
  - u2.p is equal to the pre-transfer u.p,
  - u.p is equal to nullptr, and
  - if the pre-transfer u.d maintained state, such state has been transferred to u2.d.

As in the case of a reset, u2 must properly dispose of its pre-transfer owned object via the pre-transfer associated deleter before the ownership transfer is considered complete. [Note: A deleter's state need never be copied, only moved or swapped as ownership is transferred. —end note]

- Each object of a type U instantiated from the unique\_ptr template specified in this subclause has the strict ownership semantics, specified above, of a unique pointer. In partial satisfaction of these semantics, each such U is MoveConstructible and MoveAssignable, but is not CopyConstructible nor CopyAssignable. The template parameter T of unique\_ptr may be an incomplete type.
- [Note: The uses of unique\_ptr include providing exception safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. end note]

```
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) noexcept;
  template < class T1, class D1, class T2, class D2>
    bool operator == (const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template < class T1, class D1, class T2, class D2>
    bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template < class T1, class D1, class T2, class D2>
    bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template < class T1, class D1, class T2, class D2>
    bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template < class T1, class D1, class T2, class D2>
    bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template < class T1, class D1, class T2, class D2>
    bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
 template <class T, class D>
    bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
```

§ 20.7.1 513

```
template <class T, class D>
         bool operator == (nullptr_t, const unique_ptr<T, D>& y) noexcept;
       template <class T, class D>
         bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;
       template <class T, class D>
         bool operator!=(nullptr_t, const unique_ptr<T, D>& y) noexcept;
       template <class T, class D>
         bool operator<(const unique_ptr<T, D>& x, nullptr_t);
       template <class T, class D>
         bool operator<(nullptr_t, const unique_ptr<T, D>& y);
       template <class T, class D>
         bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
       template <class T, class D>
         bool operator<=(nullptr_t, const unique_ptr<T, D>& y);
       template <class T, class D>
         bool operator>(const unique_ptr<T, D>& x, nullptr_t);
       template <class T, class D>
         bool operator>(nullptr_t, const unique_ptr<T, D>& y);
       template <class T, class D>
         bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
       template <class T, class D>
         bool operator>=(nullptr_t, const unique_ptr<T, D>& y);
    }
   20.7.1.1 Default deleters
                                                                                        [unique.ptr.dltr]
   20.7.1.1.1 In general
                                                                               [unique.ptr.dltr.general]
<sup>1</sup> The class template default_delete serves as the default deleter (destruction policy) for the class template
   unique_ptr.
<sup>2</sup> The template parameter T of default_delete may be an incomplete type.
                                                                                   [unique.ptr.dltr.dflt]
   20.7.1.1.2 default_delete
     namespace std {
       template <class T> struct default_delete {
         constexpr default_delete() noexcept = default;
         template <class U> default_delete(const default_delete<U>&) noexcept;
         void operator()(T*) const;
       };
     }
   template <class U> default_delete(const default_delete<U>& other) noexcept;
        Effects: Constructs a default_delete object from another default_delete<U> object.
        Remarks: This constructor shall not participate in overload resolution unless U* is implicitly convertible
        to T*.
   void operator()(T *ptr) const;
        Effects: calls delete on ptr.
        Remarks: If T is an incomplete type, the program is ill-formed.
```

§ 20.7.1.1.2 514

1

3

4

```
20.7.1.1.3 default_delete<T[]>
                                                                              [unique.ptr.dltr.dflt1]
 namespace std {
    template <class T> struct default_delete<T[]> {
      constexpr default_delete() noexcept = default;
      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.
20.7.1.2 unique_ptr for single objects
                                                                                  [unique.ptr.single]
 namespace std {
    template <class T, class D = default_delete<T>> class unique_ptr {
      typedef see below pointer;
      typedef T element_type;
      typedef D deleter_type;
      // 20.7.1.2.1, constructors
      constexpr unique_ptr() noexcept;
      explicit unique_ptr(pointer p) noexcept;
      unique_ptr(pointer p, see below d1) noexcept;
      unique_ptr(pointer p, see below d2) noexcept;
      unique_ptr(unique_ptr&& u) noexcept;
      constexpr unique_ptr(nullptr_t) noexcept
        : unique_ptr() { }
      template <class U, class E>
        unique_ptr(unique_ptr<U, E>&& u) noexcept;
      template <class U>
        unique_ptr(auto_ptr<U>&& u) noexcept;
      // 20.7.1.2.2, destructor
      ~unique_ptr();
      // 20.7.1.2.3, assignment
      unique_ptr& operator=(unique_ptr&& u) noexcept;
      template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
      unique_ptr& operator=(nullptr_t) noexcept;
      // 20.7.1.2.4, observers
      typename add_lvalue_reference<T>::type operator*() const;
      pointer operator->() const noexcept;
      pointer get() const noexcept;
      deleter_type& get_deleter() noexcept;
      const deleter_type& get_deleter() const noexcept;
      explicit operator bool() const noexcept;
      // 20.7.1.2.5 modifiers
      pointer release() noexcept;
      void reset(pointer p = pointer()) noexcept;
```

§ 20.7.1.2 515

```
void swap(unique_ptr& u) noexcept;

// disable copy from lvalue
unique_ptr(const unique_ptr&) = delete;
unique_ptr& operator=(const unique_ptr&) = delete;
};
}
```

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.

- <sup>2</sup> If the deleter's type D is not a reference type, D shall satisfy the requirements of Destructible (Table 24).
- 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 (17.6.3.3).
- <sup>4</sup> [Example: Given an allocator type X (17.6.3.5) and letting A be a synonym for allocator\_traits<X>, the types A::pointer, A::const\_pointer, A::void\_pointer, and A::const\_void\_pointer may be used as unique\_ptr<T, D>::pointer. end example]

```
20.7.1.2.1 unique_ptr constructors
```

[unique.ptr.single.ctor]

constexpr unique\_ptr() noexcept;

1

- Requires: D shall satisfy the requirements of DefaultConstructible (Table 19), and that construction shall not throw an exception.
- 2 Effects: Constructs a unique\_ptr object that owns nothing, value-initializing the stored pointer and the stored deleter.
- 3 Postconditions: get() == nullptr. get\_deleter() returns a reference to the stored deleter.
- 4 Remarks: If this constructor is instantiated with a pointer type or reference type for the template argument D, the program is ill-formed.

```
explicit unique_ptr(pointer p) noexcept;
```

- Requires: D shall satisfy the requirements of DefaultConstructible (Table 19), and that construction shall not throw an exception.
- 6 Effects: Constructs a unique\_ptr which owns p, initializing the stored pointer with p and value-initializing the stored deleter.
- Postconditions: get() == p. get\_deleter() returns a reference to the stored deleter.
- <sup>8</sup> Remarks: If this constructor is instantiated with a pointer type or reference type for the template argument D, the program is ill-formed.

```
unique_ptr(pointer p, see below d1) noexcept;
unique_ptr(pointer p, see below d2) noexcept;
```

The signature of these constructors depends upon whether D is a reference type. If D is 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:

§ 20.7.1.2.1 516

```
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);
```

12 Requires:

11

- If D is not an lvalue-reference type then
  - If d is an Ivalue or const rvalue then the first constructor of this pair will be selected. D shall satisfy the requirements of CopyConstructible (Table 21), 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 20), 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 Ivalue-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 Ivalue 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 Ivalue 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.

```
[Example:
```

- end example]

unique\_ptr(unique\_ptr&& u) noexcept;

- Requires: If D is not a reference type, D shall satisfy the requirements of MoveConstructible (Table 20). 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.

§ 20.7.1.2.1 517

```
template <class U, class E> unique_ptr(unique_ptr<U, E>&& u) noexcept;
```

Requires: If E is not a reference type, construction of the deleter from an rvalue of type E shall be well formed and shall not throw an exception. Otherwise, E is a reference type and construction of the deleter from an lvalue of type E shall be well formed and shall not throw an exception.

- 19 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().

```
template <class U>
  unique_ptr(auto_ptr<U>&& u) noexcept;
```

- 22 Effects: Constructs a unique\_ptr object, initializing the stored pointer with u.release() and valueinitializing 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.
- Remarks: This constructor shall not participate in overload resolution unless U\* is implicitly convertible to T\* and D is the same type as default\_delete<T>.

# 20.7.1.2.2 unique\_ptr destructor

[unique.ptr.single.dtor]

~unique\_ptr();

1

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()).

#### 20.7.1.2.3 unique\_ptr assignment

[unique.ptr.single.asgn]

unique\_ptr& operator=(unique\_ptr&& u) noexcept;

- Requires: If D is not a reference type, D shall satisfy the requirements of MoveAssignable (Table 22) 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.
- 2 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()).
- 3 Returns: \*this.

template <class U, class E> unique\_ptr& operator=(unique\_ptr<U, E>&& u) noexcept;

§ 20.7.1.2.3 518

Requires: If E is not a reference type, assignment of the deleter from an rvalue of type E shall be

4

```
well-formed and shall not throw an exception. Otherwise, E is a reference type and assignment of the
         deleter from an Ivalue of type E shall be well-formed and shall not throw an exception.
5
         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.
6
         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()).
7
         Returns: *this.
   unique_ptr& operator=(nullptr_t) noexcept;
8
         Effects: reset().
         Postcondition: get() == nullptr
9
10
         Returns: *this.
                                                                            [unique.ptr.single.observers]
   20.7.1.2.4 unique_ptr observers
   typename add_lvalue_reference<T>::type operator*() const;
1
         Requires: get() != nullptr.
2
         Returns: *get().
   pointer operator->() const noexcept;
3
         Requires: get() != nullptr.
         Returns: get().
4
5
         Note: use typically requires that T be a complete type.
   pointer get() const noexcept;
6
         Returns: The stored pointer.
   deleter_type& get_deleter() noexcept;
   const deleter_type& get_deleter() const noexcept;
7
         Returns: A reference to the stored deleter.
   explicit operator bool() const noexcept;
         Returns: get() != nullptr.
```

§ 20.7.1.2.4 519

```
20.7.1.2.5 unique_ptr modifiers
                                                                           [unique.ptr.single.modifiers]
   pointer release() noexcept;
1
        Postcondition: get() == nullptr.
2
        Returns: The value get() had at the start of the call to release.
   void reset(pointer p = pointer()) noexcept;
3
        Requires: The expression get_deleter()(get()) shall be well formed, shall have well-defined behav-
        ior, and shall not throw exceptions.
4
        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]
5
        Postconditions: get() == p. [Note: The postcondition does not hold if the call to get_deleter()
        destroys *this since this->get() is no longer a valid expression. — end note
   void swap(unique_ptr& u) noexcept;
6
        Requires: get_deleter() shall be swappable (17.6.3.2) and shall not throw an exception under swap.
7
        Effects: Invokes swap on the stored pointers and on the stored deleters of *this and u.
                                                                                   [unique.ptr.runtime]
   20.7.1.3 unique_ptr for array objects with a runtime length
     namespace std {
       template <class T, class D> class unique_ptr<T[], D> {
         typedef see below pointer;
         typedef T element_type;
         typedef D deleter_type;
         // 20.7.1.3.1, constructors
         constexpr unique_ptr() noexcept;
         explicit unique_ptr(pointer p) noexcept;
         unique_ptr(pointer p, see below d) noexcept;
         unique_ptr(pointer p, see below d) noexcept;
         unique_ptr(unique_ptr&& u) noexcept;
         constexpr unique_ptr(nullptr_t) noexcept : unique_ptr() { }
         // destructor
         ~unique_ptr();
         // assignment
         unique_ptr& operator=(unique_ptr&& u) noexcept;
         unique_ptr& operator=(nullptr_t) noexcept;
         // 20.7.1.3.2, observers
         T& operator[](size_t i) const;
         pointer get() const noexcept;
         deleter_type& get_deleter() noexcept;
         const deleter_type& get_deleter() const noexcept;
         explicit operator bool() const noexcept;
```

§ 20.7.1.3 520

```
// 20.7.1.3.3 modifiers
    pointer release() noexcept;
    void reset(pointer p = pointer()) noexcept;
    void reset(nullptr_t) noexcept;
    template <class U> void reset(U) = delete;
    void swap(unique_ptr& u) noexcept;
    // disable copy from lvalue
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
  };
}
```

- <sup>1</sup> A specialization for array types is provided with a slightly altered interface.
  - Conversions between 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 [].
- <sup>2</sup> Descriptions are provided below only for member functions that have behavior different from the primary
- The template argument T shall be a complete type.

### 20.7.1.3.1 unique\_ptr constructors

[unique.ptr.runtime.ctor]

```
explicit unique_ptr(pointer p) noexcept;
unique_ptr(pointer p, see below d) noexcept;
unique_ptr(pointer p, see below d) noexcept;
```

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]

# 20.7.1.3.2 unique\_ptr observers

[unique.ptr.runtime.observers]

```
T& operator[](size_t i) const;
```

Requires: i < the number of elements in the array to which the stored pointer points.

2 Returns: get()[i].

1

1

2

### 20.7.1.3.3 unique\_ptr modifiers

[unique.ptr.runtime.modifiers]

```
void reset(pointer p = pointer()) noexcept;
void reset(nullptr_t p) noexcept;
     Effects: If get() == nullptr there are no effects. Otherwise get_deleter()(get()).
     Postcondition: get() == p.
```

§ 20.7.1.3.3 521

```
20.7.1.4 unique_ptr specialized algorithms
                                                                                    [unique.ptr.special]
   template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;
         Effects: Calls x.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.get() == y.get().
   template <class T1, class D1, class T2, class D2>
     bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
         Returns: x.get() != y.get().
   template <class T1, class D1, class T2, class D2>
     bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
4
         Requires: Let CT be common_type<unique_ptr<T1, D1>::pointer, unique_ptr<T2, D2>::poin-
         ter>::type. Then the specialization less<CT> shall be a function object type (20.8) that induces a
         strict weak ordering (25.4) on the pointer values.
5
         Returns: less<CT>()(x.get(), y.get()).
6
         Remarks: If unique_ptr<T1, D1>::pointer is not implicitly convertible to CT or unique_ptr<T2,
         D2>::pointer is not implicitly convertible to CT, the program is ill-formed.
   template <class T1, class D1, class T2, class D2>
     bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
         Returns: !(y < x).
   template <class T1, class D1, class T2, class D2>
     bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
         Returns: y < x.
   template <class T1, class D1, class T2, class D2>
     bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
         Returns: !(x < y).
   template <class T, class D>
     bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
   template <class T, class D>
     bool operator==(nullptr_t, const unique_ptr<T, D>& x) noexcept;
10
         Returns: !x.
```

§ 20.7.1.4 522

```
template <class T, class D>
     bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;
   template <class T, class D>
     bool operator!=(nullptr_t, const unique_ptr<T, D>& x) noexcept;
         Returns: (bool)x.
   template <class T, class D>
     bool operator<(const unique_ptr<T, D>& x, nullptr_t);
   template <class T, class D>
     bool operator<(nullptr_t, const unique_ptr<T, D>& x);
12
         Requires: The specialization less<unique_ptr<T, D>::pointer> shall be a function object type (20.8)
         that induces a strict weak ordering (25.4) on the pointer values.
13
         Returns: The first function template returns less<unique_ptr<T, D>::pointer>()(x.get(),
         nullptr). The second function template returns less<unique_ptr<T, D>::pointer>()(nullptr,
         x.get()).
   template <class T, class D>
     bool operator>(const unique_ptr<T, D>& x, nullptr_t);
   template <class T, class D>
      bool operator>(nullptr_t, const unique_ptr<T, D>& x);
14
         Returns: The first function template returns nullptr < x. The second function template returns x <
         nullptr.
   template <class T, class D>
     bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
   template <class T, class D>
     bool operator<=(nullptr_t, const unique_ptr<T, D>& x);
15
         Returns: The first function template returns ! (nullptr < x). The second function template returns
         !(x < nullptr).
   template <class T, class D>
     bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
   template <class T, class D>
     bool operator>=(nullptr_t, const unique_ptr<T, D>& x);
16
         Returns: The first function template returns !(x < nullptr). The second function template returns
         !(nullptr < x).
   20.7.2
             Shared-ownership pointers
                                                                                        [util.smartptr]
   20.7.2.1
              Class bad_weak_ptr
                                                                                [util.smartptr.weakptr]
     namespace std {
       class bad_weak_ptr: public std::exception {
       public:
         bad_weak_ptr() noexcept;
     } // namespace std
   § 20.7.2.1
                                                                                                      523
```

An exception of type bad\_weak\_ptr is thrown by the shared\_ptr constructor taking a weak\_ptr.

```
bad_weak_ptr() noexcept;
    Postconditions: what() returns "bad_weak_ptr".
```

### 20.7.2.2 Class template shared\_ptr

# [util.smartptr.shared]

The shared\_ptr class template stores a pointer, usually obtained via new. shared\_ptr implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. A shared\_ptr object is empty if it does not own a pointer.

```
namespace std {
  template<class T> class shared_ptr {
  public:
    typedef T element_type;
    // 20.7.2.2.1, constructors:
    constexpr shared_ptr() noexcept;
    template<class Y> explicit shared_ptr(Y* p);
    template<class Y, class D> shared_ptr(Y* p, D d);
    template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
    template <class D> shared_ptr(nullptr_t p, D d)
    template <class D, class A> shared_ptr(nullptr_t p, D d, A a)
    template<class Y> shared_ptr(const shared_ptr<Y>& r, T *p) noexcept;
    shared_ptr(const shared_ptr& r) noexcept;
    template<class Y> shared_ptr(const shared_ptr<Y>& r) noexcept;
    shared_ptr(shared_ptr&& r) noexcept;
    template<class Y> shared_ptr(shared_ptr<Y>&& r) noexcept;
    template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);
    template<class Y> shared_ptr(auto_ptr<Y>&& r);
    template <class Y, class D> shared_ptr(unique_ptr<Y, D>&& r);
    constexpr shared_ptr(nullptr_t) : shared_ptr() { }
    // 20.7.2.2.2, destructor:
    ~shared_ptr();
    // 20.7.2.2.3, assignment:
    shared_ptr& operator=(const shared_ptr& r) noexcept;
    template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;
    shared_ptr& operator=(shared_ptr&& r) noexcept;
    template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
    template<class Y> shared_ptr& operator=(auto_ptr<Y>&& r);
    template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);
    // 20.7.2.2.4, modifiers:
    void swap(shared_ptr& r) noexcept;
    void reset() noexcept;
    template<class Y> void reset(Y* p);
    template<class Y, class D> void reset(Y* p, D d);
    template < class Y, class D, class A > void reset(Y* p, D d, A a);
    // 20.7.2.2.5, observers:
    T* get() const noexcept;
    T& operator*() const noexcept;
    T* operator->() const noexcept;
```

§ 20.7.2.2 524

```
long use_count() const noexcept;
  bool unique() const noexcept;
  explicit operator bool() const noexcept;
  template<class U> bool owner_before(shared_ptr<U> const& b) const;
  template<class U> bool owner_before(weak_ptr<U> const& b) const;
};
// 20.7.2.2.6, shared ptr creation
template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);
// 20.7.2.2.7, shared_ptr comparisons:
template < class T, class U>
  bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template < class T, class U>
  bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template < class T, class U>
  bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template < class T, class U>
  bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template < class T, class U>
  bool operator>=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template <class T>
  bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator==(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator!=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator<(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator<=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator<=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator>(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& b) noexcept;
// 20.7.2.2.8, shared_ptr specialized algorithms:
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;
// 20.7.2.2.9, shared_ptr casts:
```

§ 20.7.2.2 525

```
template < class T, class U>
         shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
       template < class T, class U>
         shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
       template < class T, class U>
         shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
       // 20.7.2.2.10, shared ptr get deleter:
       template<class D, class T> D* get_deleter(const shared_ptr<T>& p) noexcept;
       // 20.7.2.2.11, shared_ptr I/O:
       template < class E, class T, class Y>
         basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>& p);
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]
<sup>4</sup> 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.
   20.7.2.2.1 shared_ptr constructors
                                                                           [util.smartptr.shared.const]
   constexpr shared_ptr() noexcept;
1
        Effects: Constructs an empty shared_ptr object.
2
        Postconditions: use_count() == 0 && get() == 0.
   template<class Y> explicit shared_ptr(Y* p);
3
        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.
4
        Effects: Constructs a shared_ptr object that owns the pointer p.
5
        Postconditions: use_count() == 1 && get() == p.
6
        Throws: bad_alloc, or an implementation-defined exception when a resource other than memory
        could not be obtained.
7
        Exception safety: If an exception is thrown, delete p is called.
   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);
```

§ 20.7.2.2.1 526

```
Requires: p shall be convertible to T*. D shall be CopyConstructible. The copy constructor and destructor of D shall not throw exceptions. The expression d(p) shall be well formed, shall have well defined behavior, and shall not throw exceptions. A shall be an allocator (17.6.3.5). The copy constructor and destructor of A shall not throw exceptions.
```

- 9 Effects: Constructs a shared\_ptr object that owns the object p and the deleter d. The second and fourth constructors shall use a copy of a to allocate memory for internal use.
- 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) noexcept;

- 13 Effects: Constructs a shared\_ptr instance that stores p and shares ownership with r.
- Postconditions: get() == p && use\_count() == r.use\_count()
- [Note: To avoid the possibility of a dangling pointer, the user of this constructor 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) noexcept; template<class Y> shared\_ptr(const shared\_ptr<Y>& r) noexcept;

- 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.
- 19 Postconditions: get() == r.get() && use\_count() == r.use\_count().

```
shared_ptr(shared_ptr&& r) noexcept;
template<class Y> shared_ptr(shared_ptr<Y>&& r) noexcept;
```

- Remark: The second constructor shall not participate in overload resolution unless Y\* is convertible to T\*.
- 21 Effects: Move-constructs a shared\_ptr instance from r.
- Postconditions: \*this shall contain the old value of r. r shall be empty. r.get() == 0.

#### template<class Y> explicit shared\_ptr(const weak\_ptr<Y>& r);

- 23 Requires: Y\* shall be convertible to T\*.
- 24 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().
- 27 Exception safety: If an exception is thrown, the constructor has no effect.

§ 20.7.2.2.1 527

```
template<class Y> shared_ptr(auto_ptr<Y>&& r);
28
         Requires: r.release() shall be convertible to T*. Y shall be a complete type. The expression delete
         r.release() shall be well formed, shall have well defined behavior, and shall not throw exceptions.
29
         Effects: Constructs a shared ptr object that stores and owns r.release().
30
         Postconditions: use_count() == 1 && r.get() == 0.
31
         Throws: bad_alloc, or an implementation-defined exception when a resource other than memory
         could not be obtained.
32
         Exception safety: If an exception is thrown, the constructor has no effect.
   template <class Y, class D> shared_ptr(unique_ptr<Y, D>&&r);
33
         Effects: Equivalent to shared_ptr(r.release(), r.get_deleter()) when D is not a reference type,
         otherwise shared_ptr(r.release(), ref(r.get_deleter())).
34
         Exception safety: If an exception is thrown, the constructor has no effect.
   20.7.2.2.2 shared ptr destructor
                                                                              [util.smartptr.shared.dest]
    ~shared_ptr();
1
         Effects:
          — If *this is empty or shares ownership with another shared_ptr instance (use_count() > 1),
              there are no side effects.
           — Otherwise, if *this owns an object p and a deleter d, d(p) is called.
           — Otherwise, *this owns a pointer p, and delete p is called.
  [Note: Since the destruction of *this decreases the number of instances that share ownership with *this
   by one, after *this has been destroyed all shared_ptr instances that shared ownership with *this will
   report a use_count() that is one less than its previous value. — end note]
                                                                            [util.smartptr.shared.assign]
   20.7.2.2.3
                shared_ptr assignment
   shared_ptr& operator=(const shared_ptr& r) noexcept;
   template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;
    template<class Y> shared_ptr& operator=(auto_ptr<Y>&& r);
1
         Effects: Equivalent to shared_ptr(r).swap(*this).
2
         Returns: *this.
3
         [Note: The use count updates caused by the temporary object construction and destruction are not
         observable side effects, so the implementation may meet the effects (and the implied guarantees) via
         different means, without creating a temporary. In particular, in the example:
           shared_ptr<int> p(new int);
           shared_ptr<void> q(p);
           p = p;
           q = p;
         both assignments may be no-ops. — end note]
   shared_ptr& operator=(shared_ptr&& r) noexcept;
   template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
   § 20.7.2.2.3
                                                                                                       528
```

```
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
   20.7.2.2.4 shared_ptr modifiers
                                                                            [util.smartptr.shared.mod]
   void swap(shared_ptr& r) noexcept;
        Effects: Exchanges the contents of *this and r.
   void reset() noexcept;
2
        Effects: Equivalent to shared_ptr().swap(*this).
   template<class Y> void reset(Y* p);
3
        Effects: Equivalent to shared_ptr(p).swap(*this).
   template<class Y, class D> void reset(Y* p, D d);
4
        Effects: Equivalent to shared_ptr(p, d).swap(*this).
   template<class Y, class D, class A> void reset(Y* p, D d, A a);
5
        Effects: Equivalent to shared_ptr(p, d, a).swap(*this).
   20.7.2.2.5
              shared_ptr observers
                                                                             [util.smartptr.shared.obs]
   T* get() const noexcept;
        Returns: the stored pointer.
   T& operator*() const noexcept;
2
        Requires: get() != 0.
3
        Returns: *get().
4
        Remarks: When T is void, it is unspecified whether this member function is declared. If it is declared,
        it is unspecified what its return type is, except that the declaration (although not necessarily the
        definition) of the function shall be well formed.
   T* operator->() const noexcept;
5
        Requires: get() != 0.
6
        Returns: get().
```

§ 20.7.2.2.5 529

```
long use count() const noexcept;
         Returns: the number of shared_ptr objects, *this included, that share ownership with *this, or 0
         when *this is empty.
8
         [Note: use count() is not necessarily efficient. — end note]
   bool unique() const noexcept;
9
         Returns: use_count() == 1.
10
         [Note: unique() may be faster than use_count(). If you are using unique() to implement copy on
         write, do not rely on a specific value when get() == 0. — end note]
   explicit operator bool() const noexcept;
11
         Returns: get() != 0.
   template<class U> bool owner_before(shared_ptr<U> const& b) const;
   template<class U> bool owner_before(weak_ptr<U> const& b) const;
12
         Returns: An unspecified value such that
           x.owner_before(y) defines a strict weak ordering as defined in 25.4;
          — 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 own-
              ership or are both empty.
   20.7.2.2.6
                shared_ptr creation
                                                                           [util.smartptr.shared.create]
   template < class T, class... Args > shared_ptr < T > make_shared(Args&&... args);
   template < class T, class A, class... Args>
      shared_ptr<T> allocate_shared(const A& a, Args&&... args);
         Requires: The expression ::new (pv) T(std::forward<Args>(args)...), where pv has type void*
         and points to storage suitable to hold an object of type T, shall be well formed. A shall be an alloca-
         tor (17.6.3.5). The copy constructor and destructor of A shall not throw exceptions.
2
         Effects: Allocates memory suitable for an object of type T and constructs an object in that memory
         via the placement new expression ::new (pv) T(std::forward<Args>(args)...). The template
         allocate_shared uses a copy of a to allocate memory. If an exception is thrown, the functions have
         no effect.
3
         Returns: A shared_ptr instance that stores and owns the address of the newly constructed object of
         type T.
4
         Postconditions: get() != 0 && use count() == 1
5
         Throws: bad alloc, or an exception thrown from A::allocate or from the constructor of T.
6
         Remarks: Implementations are encouraged, but not required, to perform no more than one memory
         allocation. [Note: This provides efficiency equivalent to an intrusive smart pointer. — end note]
         Note: These functions will typically allocate more memory than sizeof(T) to allow for internal
         bookkeeping structures such as the reference counts. — end note]
```

§ 20.7.2.2.6 530

[util.smartptr.shared.cmp]

20.7.2.2.7 shared\_ptr comparison

```
template<class T, class U> bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
1
        Returns: a.get() == b.get().
   template<class T, class U> bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
2
        Returns: less<V>()(a.get(), b.get()), where V is the composite pointer type (5.9) of T* and U*.
3
        [Note: Defining a comparison operator allows shared_ptr objects to be used as keys in associative
        containers. — end note
   template <class T>
     bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
   template <class T>
     bool operator==(nullptr_t, const shared_ptr<T>& a) noexcept;
        Returns: !a.
   template <class T>
     bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;
   template <class T>
     bool operator!=(nullptr_t, const shared_ptr<T>& a) noexcept;
        Returns: (bool)a.
   template <class T>
     bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;
   template <class T>
     bool operator<(nullptr_t, const shared_ptr<T>& a) noexcept;
6
        Returns: The first function template returns less<T*>()(a.get(), nullptr). The second function
        template returns less<T*>()(nullptr, a.get()).
   template <class T>
     bool operator>(const shared_ptr<T>& a, nullptr_t) noexcept;
   template <class T>
     bool operator>(nullptr_t, const shared_ptr<T>& a) noexcept;
7
        Returns: The first function template returns nullptr < a. The second function template returns a <
        nullptr.
   template <class T>
     bool operator<=(const shared_ptr<T>& a, nullptr_t) noexcept;
   template <class T>
     bool operator<=(nullptr_t, const shared_ptr<T>& a) noexcept;
        Returns: The first function template returns ! (nullptr < a). The second function template returns
        !(a < nullptr).
```

§ 20.7.2.2.7 531

template <class T>

1

1

2

3

4

5

6

7

8

9

10

11

12

```
bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
 bool operator>=(nullptr_t, const shared_ptr<T>& a) noexcept;
     Returns: The first function template returns ! (a < nullptr). The second function template returns
     !(nullptr < a).
20.7.2.2.8 shared_ptr specialized algorithms
                                                                        [util.smartptr.shared.spec]
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;
     Effects: Equivalent to a.swap(b).
20.7.2.2.9 shared_ptr casts
                                                                        [util.smartptr.shared.cast]
template<class T, class U> shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
     Requires: The expression static cast<T*>(r.get()) shall be well formed.
     Returns: If r is empty, an empty shared ptr<T>; otherwise, a shared ptr<T> object that stores
     static_cast<T*>(r.get()) and shares ownership with r.
     Postconditions: w.get() == static_cast<T*>(r.get()) and w.use_count() == r.use_count(),
     where w is the return value.
     [Note: The seemingly equivalent expression shared_ptr<T>(static_cast<T*>(r.get())) will even-
     tually result in undefined behavior, attempting to delete the same object twice. — end note
template<class T, class U> shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
     Requires: The expression dynamic_cast<T*>(r.get()) shall be well formed and shall have well defined
     behavior.
     Returns:
      — When dynamic_cast<T*>(r.get()) returns a nonzero value, a shared_ptr<T> object that stores
          a copy of it and shares ownership with r;
      — Otherwise, an empty shared_ptr<T> object.
     Postcondition: w.get() == dynamic_cast<T*>(r.get()), where w is the return value.
     [Note: The seemingly equivalent expression shared_ptr<T>(dynamic_cast<T*>(r.get())) will even-
     tually result in undefined behavior, attempting to delete the same object twice. -end note
template<class T, class U> shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
     Requires: The expression const_cast<T*>(r.get()) shall be well formed.
     Returns: If r is empty, an empty shared_ptr<T>; otherwise, a shared_ptr<T> object that stores
     const_cast<T*>(r.get()) and shares ownership with r.
     Postconditions: w.get() == const_cast<T*>(r.get()) and w.use_count() == r.use_count(), where
     w is the return value.
     [Note: The seemingly equivalent expression shared_ptr<T>(const_cast<T*>(r.get())) will even-
     tually result in undefined behavior, attempting to delete the same object twice. — end note
```

§ 20.7.2.2.9 532

```
20.7.2.2.10 get_deleter
```

1

[util.smartptr.getdeleter]

template<class D, class T> D\* get\_deleter(const shared\_ptr<T>& p) noexcept;

Returns: If p owns a deleter d of type cv-unqualified D, returns &d; otherwise returns 0. The returned pointer remains valid as long as there exists a shared\_ptr instance that owns d. [Note: It is unspecified whether the pointer remains valid longer than that. This can happen if the implementation doesn't destroy the deleter until all weak\_ptr instances that share ownership with p have been destroyed. — end note]

```
20.7.2.2.11 shared_ptr I/O
```

[util.smartptr.shared.io]

```
template<class E, class T, class Y>
    basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);

Effects: os << p.get();.

Returns: os.</pre>
```

## 20.7.2.3 Class template weak\_ptr

[util.smartptr.weak]

The weak\_ptr class template stores a weak reference to an object that is already managed by a shared\_ptr. To access the object, a weak\_ptr can be converted to a shared\_ptr using the member function lock.

```
namespace std {
  template<class T> class weak_ptr {
  public:
    typedef T element_type;
    // 20.7.2.3.1, constructors
    constexpr weak_ptr() noexcept;
    template<class Y> weak_ptr(shared_ptr<Y> const& r) noexcept;
    weak_ptr(weak_ptr const& r) noexcept;
    template<class Y> weak_ptr(weak_ptr<Y> const& r) noexcept;
    // 20.7.2.3.2, destructor
    ~weak_ptr();
    // 20.7.2.3.3, assignment
    weak_ptr& operator=(weak_ptr const& r) noexcept;
    template<class Y> weak_ptr& operator=(weak_ptr<Y> const& r) noexcept;
    template<class Y> weak_ptr& operator=(shared_ptr<Y> const& r) noexcept;
    // 20.7.2.3.4, modifiers
    void swap(weak_ptr& r) noexcept;
    void reset() noexcept;
    // 20.7.2.3.5, observers
    long use_count() const noexcept;
    bool expired() const noexcept;
    shared_ptr<T> lock() const noexcept;
    template<class U> bool owner_before(shared_ptr<U> const& b);
    template<class U> bool owner_before(weak_ptr<U> const& b);
  };
  // 20.7.2.3.6, specialized algorithms
  template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
} // namespace std
```

§ 20.7.2.3 533

```
Specializations of weak_ptr shall be CopyConstructible and CopyAssignable, allowing their use in stan-
   dard containers. The template parameter T of weak_ptr may be an incomplete type.
                                                                             [util.smartptr.weak.const]
   20.7.2.3.1 weak_ptr constructors
   constexpr weak_ptr() noexcept;
1
        Effects: Constructs an empty weak_ptr object.
2
        Postconditions: use_count() == 0.
   weak_ptr(const weak_ptr& r) noexcept;
   template<class Y> weak_ptr(const weak_ptr<Y>& r) noexcept;
   template<class Y> weak_ptr(const shared_ptr<Y>& r) noexcept;
3
        Requires: The second and third constructors shall not participate in the overload resolution unless Y*
        is implicitly convertible to T*.
4
        Effects: If r is empty, constructs an empty weak_ptr object; otherwise, constructs a weak_ptr object
        that shares ownership with r and stores a copy of the pointer stored in r.
        Postconditions: use_count() == r.use_count().
   20.7.2.3.2 weak ptr destructor
                                                                              [util.smartptr.weak.dest]
   ~weak_ptr();
        Effects: Destroys this weak_ptr object but has no effect on the object its stored pointer points to.
   20.7.2.3.3 weak_ptr assignment
                                                                            [util.smartptr.weak.assign]
   weak_ptr& operator=(const weak_ptr& r) noexcept;
   template<class Y> weak_ptr& operator=(const weak_ptr<Y>& r) noexcept;
   template<class Y> weak_ptr& operator=(const shared_ptr<Y>& r) noexcept;
1
        Effects: Equivalent to weak_ptr(r).swap(*this).
2
        Remarks: The implementation may meet the effects (and the implied guarantees) via different means,
        without creating a temporary.
                                                                              [util.smartptr.weak.mod]
   20.7.2.3.4 weak_ptr modifiers
   void swap(weak_ptr& r) noexcept;
        Effects: Exchanges the contents of *this and r.
   void reset() noexcept;
        Effects: Equivalent to weak_ptr().swap(*this).
   20.7.2.3.5 weak ptr observers
                                                                               [util.smartptr.weak.obs]
   long use_count() const noexcept;
1
        Returns: 0 if *this is empty; otherwise, the number of shared_ptr instances that share ownership
        with *this.
2
        [Note: use_count() is not necessarily efficient. — end note]
   bool expired() const noexcept;
   § 20.7.2.3.5
                                                                                                      534
```

```
Returns: use\_count() == 0.
4
        [Note: expired() may be faster than use_count(). — end note]
   shared_ptr<T> lock() const noexcept;
        Returns: expired() ? shared_ptr<T>() : shared_ptr<T>(*this).
   template<class U> bool owner_before(shared_ptr<U> const& b);
   template<class U> bool owner_before(weak_ptr<U> const& b);
        Returns: An unspecified value such that

    x.owner_before(y) defines a strict weak ordering as defined in 25.4;

         — 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 own-
             ership or are both empty.
   20.7.2.3.6 weak_ptr specialized algorithms
                                                                             [util.smartptr.weak.spec]
   template < class T > void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
        Effects: Equivalent to a.swap(b).
   20.7.2.3.7 Class template owner_less
                                                                             [util.smartptr.ownerless]
  The class template owner_less allows ownership-based mixed comparisons of shared and weak pointers.
     namespace std {
      template<class T> struct owner_less;
      template<class T> struct owner_less<shared_ptr<T> > {
         typedef bool result_type;
         typedef shared_ptr<T> first_argument_type;
         typedef shared_ptr<T> second_argument_type;
         bool operator()(shared_ptr<T> const&, shared_ptr<T> const&) const;
         bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
         bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
       };
       template<class T> struct owner_less<weak_ptr<T> > {
         typedef bool result_type;
         typedef weak_ptr<T> first_argument_type;
         typedef weak_ptr<T> second_argument_type;
         bool operator()(weak_ptr<T> const&, weak_ptr<T> const&) const;
         bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
         bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
       };
     }
  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.
```

§ 20.7.2.3.7 535

A class T can inherit from enable\_shared\_from\_this<T> to inherit the shared\_from\_this member func-

[util.smartptr.enab]

536

20.7.2.4 Class template enable\_shared\_from\_this

7

8

9

§ 20.7.2.4

```
tions that obtain a shared_ptr instance pointing to *this.
  [Example:
     struct X: public enable_shared_from_this<X> {
     };
     int main() {
       shared_ptr<X> p(new X);
       shared_ptr<X> q = p->shared_from_this();
       assert(p == q);
       assert(!(p < q) && !(q < p)); // p and q share ownership
   — end example]
     namespace std {
       template<class T> class enable_shared_from_this {
       protected:
         constexpr enable_shared_from_this() noexcept;
         enable_shared_from_this(enable_shared_from_this const&) noexcept;
         enable_shared_from_this& operator=(enable_shared_from_this const&) noexcept;
         ~enable_shared_from_this();
       public:
         shared_ptr<T> shared_from_this();
         shared_ptr<T const> shared_from_this() const;
     } // namespace std
3 The template parameter T of enable_shared_from_this may be an incomplete type.
   constexpr enable_shared_from_this() noexcept;
   enable_shared_from_this(const enable_shared_from_this<T>&) noexcept;
        Effects: Constructs an enable_shared_from_this<T> object.
   enable_shared_from_this<T>& operator=(const enable_shared_from_this<T>&) noexcept;
        Returns: *this.
   ~enable_shared_from_this();
        Effects: Destroys *this.
   shared_ptr<T>
                       shared_from_this();
   shared_ptr<T const> shared_from_this() const;
        Requires: enable_shared_from_this<T> shall be an accessible base class of T. *this shall be a
        subobject of an object t of type T. There shall be at least one shared_ptr instance p that owns &t.
        Returns: A shared_ptr<T> object r that shares ownership with p.
        Postconditions: r.get() == this.
  [Note: A possible implementation is shown below:
```

```
template<class T> class enable_shared_from_this {
     private:
        weak_ptr<T> __weak_this;
     protected:
        constexpr enable_shared_from_this() : __weak_this() { }
        enable_shared_from_this(enable_shared_from_this const &) { }
        enable_shared_from_this& operator=(enable_shared_from_this const &) { return *this; }
        ~enable_shared_from_this() { }
     public:
        shared_ptr<T> shared_from_this() { return shared_ptr<T>(__weak_this); }
        shared_ptr<T const> shared_from_this() const { return shared_ptr<T const>(_weak_this); }
11 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]
   20.7.2.5 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);
3
         Requires: p shall not be null.
4
         Returns: true if atomic access to *p is lock-free, false otherwise.
5
         Throws: Nothing.
   template<class T>
      shared_ptr<T> atomic_load(const shared_ptr<T>* p);
6
         Requires: p shall not be null.
7
         Returns: atomic_load_explicit(p, memory_order_seq_cst).
         Throws: Nothing.
   template<class T>
      shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);
9
         Requires: p shall not be null.
10
         Requires: mo shall not be memory_order_release or memory_order_acq_rel.
11
         Returns: *p.
12
         Throws: Nothing.
   template<class T>
      void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
13
         Requires: p shall not be null.
14
         Effects: atomic_store_explicit(p, r, memory_order_seq_cst).
15
         Throws: Nothing.
```

§ 20.7.2.5

```
template<class T>
      void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
16
         Requires: p shall not be null.
17
         Requires: mo shall not be memory_order_acquire or memory_order_acq_rel.
18
         Effects: p->swap(r).
19
         Throws: Nothing.
   template<class T>
      shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
20
         Requires: p shall not be null.
21
         Returns: atomic_exchange_explicit(p, r, memory_order_seq_cst).
22
         Throws: Nothing.
   template<class T>
      shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r,
                                             memory_order mo);
23
         Requires: p shall not be null.
24
         Effects: p->swap(r).
25
         Returns: The previous value of *p.
26
         Throws: Nothing.
   template<class T>
     bool atomic_compare_exchange_weak(
        shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
27
         Requires: p shall not be null.
28
         Returns: atomic_compare_exchange_weak_explicit(p, v, w, memory_order_seq_cst, memory_-
         order_seq_cst).
29
         Throws: Nothing.
   template<class T>
     bool atomic_compare_exchange_strong(
        shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
30
         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);
   § 20.7.2.5
                                                                                                      538
```

- 31 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.
- 35 Throws: Nothing.

1

- Remarks: two shared\_ptr objects are equivalent if they store the same pointer value and share ownership.
- 37 Remarks: the weak forms may fail spuriously. See 29.6.

## 20.7.2.6 Smart pointer hash support

[util.smartptr.hash]

```
\label{lem:lemplate <class T, class D> struct hash < unique_ptr < T, D> >;
```

Requires: The template specialization shall meet the requirements of class template hash (20.8.12). 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.

```
template <class T> struct hash<shared_ptr<T> >;
```

Requires: The template specialization shall meet the requirements of class template hash (20.8.12). 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()).

## 20.8 Function objects

## [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).<sup>230</sup> A function object is an object of a function object type. In the places where one would expect to pass a pointer to a function to an algorithmic template (Clause 25), the interface is specified to accept a function object. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.
- Header <functional> synopsis

```
namespace std {
    // D.8.1, base (deprecated):
    template <class Arg, class Result> struct unary_function;
    template <class Arg1, class Arg2, class Result> struct binary_function;

// 20.8.3, reference_wrapper:
    template <class T> class reference_wrapper;

template <class T> reference_wrapper<T> ref(T&) noexcept;
    template <class T> reference_wrapper<const T> cref(const T&) noexcept;
    template <class T> void ref(const T&) = delete;
    template <class T> void cref(const T&) = delete;
```

<sup>230)</sup> 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.

```
template <class T> reference_wrapper<T> ref(reference_wrapper<T>) noexcept;
template <class T> reference_wrapper<const T> cref(reference_wrapper<T>) noexcept;
// 20.8.4, 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;
// 20.8.5, 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;
// 20.8.6, logical operations:
template <class T> struct logical_and;
template <class T> struct logical_or;
template <class T> struct logical_not;
// 20.8.7, bitwise operations:
template <class T> struct bit_and;
template <class T> struct bit_or;
template <class T> struct bit_xor;
// 20.8.8, 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&);
// 20.8.9, 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&);
// D.8.2.1, adaptors (deprecated):
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));
// D.8.2.2, adaptors (deprecated):
template<class S, class T> class mem_fun_t;
template<class S, class T, class A> class mem_fun1_t;
template < class S, class T>
    mem_fun_t<S,T> mem_fun(S (T::*f)());
template < class S, class T, class A>
    mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A));
template<class S, class T> class mem_fun_ref_t;
template<class S, class T, class A> class mem_fun1_ref_t;
template<class S, class T>
    mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)());
template < class S, class T, class A>
    mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A));
template <class S, class T> class const_mem_fun_t;
template <class S, class T, class A> class const_mem_fun1_t;
template <class S, class T>
  const_mem_fun_t<S,T> mem_fun(S (T::*f)() const);
template <class S, class T, class A>
  const_mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A) const);
template <class S, class T> class const_mem_fun_ref_t;
template <class S, class T, class A> class const_mem_fun1_ref_t;
template <class S, class T>
  const_mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)() const);
template <class S, class T, class A>
  const_mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A) const);
// 20.8.10, member function adaptors:
template<class R, class T> unspecified mem_fn(R T::*);
template<class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...));
template<class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...) const);
template<class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...) volatile);
template < class R, class T, class... Args>
```

```
unspecified mem_fn(R (T::*)(Args...) const volatile);
template<class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...) &);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...) const &);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...) volatile &);
template<class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...) const volatile &);
template<class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...) &&);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...) const &&);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...) volatile &&);
template<class R, class T, class... Args>
  unspecified mem_fn(R (T::*)(Args...) const volatile &&);
// 20.8.11 polymorphic function wrappers:
class bad_function_call;
template<class> class function; // undefined
template<class R, class... ArgTypes> class function<R(ArgTypes...)>;
template < class R, class... ArgTypes>
  void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);
template < class R, class... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>&, nullptr_t);
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...)>&);
// 20.8.12, 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:

```
{\tt transform(a.begin(), a.end(), b.begin(), a.begin(), plus < double > ());}
```

— end example]

Example: To negate every element of a:

```
transform(a.begin(), a.end(), a.begin(), negate<double>());
```

— end example]

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]

- The following definitions apply to this Clause:
- A call signature is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.
- A callable type is a function object type (20.8) or a pointer to member.
- <sup>4</sup> A callable object is an object of a callable type.
- <sup>5</sup> 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.
- <sup>7</sup> A target object is the callable object held by a call wrapper.

## 20.8.2 Requirements

[func.require]

- Define INVOKE (f, t1, t2, ..., tN) as follows:
  - (t1.\*f)(t2, ..., tN) when f is a pointer to a member function of a class T and t1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;
  - ((\*t1).\*f)(t2, ..., tN) when f is a pointer to a member function of a class T and t1 is not one of the types described in the previous item;
  - t1.\*f when N == 1 and f is a pointer to member data of a class T and t1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;
  - (\*t1).\*f when N == 1 and f is a pointer to member data of a class T and t1 is not one of the types described in the previous item;
  - f(t1, t2, ..., tN) in all other cases.
- Define INVOKE (f, t1, t2, ..., tN, R) as INVOKE (f, t1, t2, ..., tN) implicitly converted to R.
- <sup>3</sup> If a call wrapper (20.8.1) has a *weak result type* the type of its member type result\_type is based on the type T of the wrapper's target object (20.8.1):
  - if T is a pointer to function type, result\_type shall be a synonym for the return type of T;
  - if T is a pointer to member function, result\_type shall be a synonym for the return type of T;

§ 20.8.2 543

— if T is a class type with a member type result\_type, then result\_type shall be a synonym for T::result\_type;

- otherwise result\_type shall not be defined.
- <sup>4</sup> 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 arbitrary argument list and delivers the arguments to the wrapped callable object as references. This forwarding step shall ensure that rvalue arguments are delivered as rvalue-references and lvalue arguments are delivered as lvalue-references. [Note: In a typical implementation forwarding call wrappers have an overloaded function call operator of the form

```
\label{lem:condargs} $$ $$ R  operator()(UnBoundArgs&&... unbound_args) $$ $ cv-qual; $$ --end  note]
```

## 20.8.3 Class template reference\_wrapper

[refwrap]

```
namespace std {
  template <class T> class reference_wrapper {
  public :
    // types
    typedef T type;
                                                   // not always defined
    typedef see below result_type;
    typedef see below argument_type;
                                                   // not always defined
                                                   // not always defined
    typedef see below first_argument_type;
    typedef see below second_argument_type;
                                                   // not always defined
    // construct/copy/destroy
    reference_wrapper(T&) noexcept;
                                          // do not bind to temporary objects
    reference_wrapper(T&&) = delete;
    reference_wrapper(const reference_wrapper<T>& x) noexcept;
    // assignment
    reference_wrapper& operator=(const reference_wrapper<T>& x) noexcept;
    // access
    operator T& () const noexcept;
    T& get() const noexcept;
    // invocation
    template <class... ArgTypes>
    typename result_of<T&(ArgTypes&&...)>::type
    operator() (ArgTypes&&...) const;
 };
```

- reference\_wrapper<T> is a CopyConstructible and CopyAssignable wrapper around a reference to an object or function of type T.
- <sup>2</sup> reference\_wrapper<T> has a weak result type (20.8.2). If T is a function type, result\_type shall be a synonym for the return type of T.
- The template instantiation reference\_wrapper<T> shall define a nested type named argument\_type as a synonym for T1 only if the type T is any of the following:
  - a function type or a pointer to function type taking one argument of type T1

```
— a pointer to member function R T0::f cv (where cv represents the member function's cv-qualifiers); the type T1 is cv T0*
```

- a class type with a member type argument\_type; the type T1 is T::argument\_type.
- The template instantiation reference\_wrapper<T> shall define two nested types named first\_argument\_type and second\_argument\_type as synonyms for T1 and T2, respectively, only if the type T is any of the following:
  - a function type or a pointer to function type taking two arguments of types T1 and T2
  - a pointer to member function R T0::f(T2) cv (where cv represents the member function's cv-qualifiers); the type T1 is cv T0\*
  - a class type with member types first\_argument\_type and second\_argument\_type; the type T1 is T::first\_argument\_type. and the type T2 is T::second\_argument\_type.

```
20.8.3.1 reference_wrapper construct/copy/destroy
```

[refwrap.const]

```
reference_wrapper(T& t) noexcept;
```

Effects: Constructs a reference\_wrapper object that stores a reference to t.

reference\_wrapper(const reference\_wrapper<T>& x) noexcept;

2 Effects: Constructs a reference\_wrapper object that stores a reference to x.get().

## 20.8.3.2 reference\_wrapper assignment

[refwrap.assign]

```
reference_wrapper& operator=(const reference_wrapper<T>& x) noexcept;

Postconditions: *this stores a reference to x.get().
```

#### 20.8.3.3 reference\_wrapper access

[refwrap.access]

```
operator T& () const noexcept; {\it Returns:} \  \, {\it The stored reference}.
```

T& get() const noexcept;

1

2 Returns: The stored reference.

## 20.8.3.4 reference\_wrapper invocation

[refwrap.invoke]

```
template <class... ArgTypes>
typename result_of<T&(ArgTypes&&...)>::type
operator()(ArgTypes&&... args) const;
Returns: INVOKE(get(), std::forward<ArgTypes>(args)...). (20.8.2)
```

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.

§ 20.8.3.4 545

```
20.8.3.5 reference_wrapper helper functions
                                                                                    [refwrap.helpers]
   template <class T> reference_wrapper<T> ref(T& t) noexcept;
1
        Returns: reference_wrapper<T>(t)
   template <class T> reference_wrapper<T> ref(reference_wrapper<T>t) noexcept;
2
        Returns: ref(t.get())
   template <class T> reference_wrapper<const T> cref(const T& t) noexcept;
3
        Returns: reference_wrapper <const T>(t)
   template <class T> reference_wrapper<const T> cref(reference_wrapper<T> t) noexcept;
        Returns: cref(t.get());
   20.8.4 Arithmetic operations
                                                                            [arithmetic.operations]
  The library provides basic function object classes for all of the arithmetic operators in the language (5.6,
   5.7).
   template <class T> struct plus {
    T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef T result_type;
  };
        operator() returns x + y.
   template <class T> struct minus {
    T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
     typedef T result_type;
   };
3
        operator() returns x - y.
   template <class T> struct multiplies {
    T operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
     typedef T result_type;
  };
4
        operator() returns x * y.
```

§ 20.8.4 546

```
template <class T> struct divides {
     T operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef T result_type;
   };
5
        operator() returns x / y.
   template <class T> struct modulus {
     T operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef T result_type;
  };
6
        operator() returns x % y.
   template <class T> struct negate {
     T operator()(const T& x) const;
     typedef T argument_type;
     typedef T result_type;
   };
        operator() returns -x.
   20.8.5 Comparisons
                                                                                        [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 {
     bool operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef bool result_type;
  };
2
        operator() returns x == y.
   template <class T> struct not_equal_to {
     bool operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef bool result_type;
   };
3
        operator() returns x != y.
   template <class T> struct greater {
     bool operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef bool result_type;
  };
   § 20.8.5
                                                                                                     547
```

```
operator() returns x > y.
   template <class T> struct less {
     bool operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef bool result_type;
        operator() returns x < y.
   template <class T> struct greater_equal {
     bool operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef bool result_type;
   };
6
        operator() returns x \ge y.
   template <class T> struct less_equal {
     bool operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef bool result_type;
   };
        operator() returns x <= y.
  For templates greater, less, greater_equal, and less_equal, the specializations for any pointer type
   yield a total order, even if the built-in operators <, >, <=, >= do not.
   20.8.6 Logical operations
                                                                                   [logical.operations]
<sup>1</sup> 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 {
     bool operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef bool result_type;
  };
2
        operator() returns x && y.
   template <class T> struct logical_or {
     bool operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef bool result_type;
  };
        operator() returns x || y.
   § 20.8.6
                                                                                                      548
```

```
template <class T> struct logical_not {
     bool operator()(const T& x) const;
     typedef T argument_type;
     typedef bool result_type;
   };
4
        operator() returns !x.
                                                                                  [bitwise.operations]
           Bitwise operations
<sup>1</sup> 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 {
     T operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef T result_type;
  };
2
        operator() returns x & y.
   template <class T> struct bit_or {
     T operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef T result_type;
   };
3
        operator() returns x | y.
   template <class T> struct bit_xor {
     T operator()(const T& x, const T& y) const;
     typedef T first_argument_type;
     typedef T second_argument_type;
     typedef T result_type;
   };
        operator() returns x ^ y.
   20.8.8 Negators
                                                                                             [negators]
  Negators not1 and not2 take a unary and a binary predicate, respectively, and return their comple-
   ments (5.3.1).
   template <class Predicate>
     class unary_negate {
   public:
     explicit unary_negate(const Predicate& pred);
     bool operator()(const typename Predicate::argument_type& x) const;
     typedef typename Predicate::argument_type argument_type;
     typedef bool result_type;
  };
2
        operator() returns !pred(x).
```

§ 20.8.8 549

```
template <class Predicate>
     unary_negate<Predicate> not1(const Predicate& pred);
3
        Returns: unary_negate<Predicate>(pred).
   template <class Predicate>
     class binary_negate {
     public:
       explicit binary_negate(const Predicate& pred);
       bool operator()(const typename Predicate::first_argument_type& x,
           const typename Predicate::second_argument_type& y) const;
     typedef typename Predicate::first_argument_type first_argument_type;
     typedef typename Predicate::second_argument_type second_argument_type;
     typedef bool result_type;
     };
4
        operator() returns !pred(x,y).
   template <class Predicate>
     binary_negate<Predicate> not2(const Predicate& pred);
        Returns: binary_negate<Predicate>(pred).
           Function template bind
                                                                                                   [bind]
<sup>1</sup> The function template bind returns an object that binds a callable object passed as an argument to additional
   arguments.
   20.8.9.1 Function object binders
                                                                                              [func.bind]
  This subclause describes a uniform mechanism for binding arguments of callable objects.
                                                                                       [func.bind.isbind]
   20.8.9.1.1 Class template is_bind_expression
     namespace std {
       template<class T> struct is_bind_expression
         : integral_constant<bool, see below> { };
<sup>1</sup> 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.
<sup>2</sup> 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>.
<sup>3</sup> 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>.
                                                                                        [func.bind.bind]
   20.8.9.1.2 Function template bind
<sup>1</sup> 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^{th} type in the template parameter back BoundArgs,
     — TiD is the type decay<Ti>::type,
```

§ 20.8.9.1.2 550

- 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.

```
template<class F, class... BoundArgs>
unspecified bind(F&& f, BoundArgs&&... bound_args);
```

- 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 == sizeof...(bound\_args).
- 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, 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.
- 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]

```
template<class R, class F, class... BoundArgs>
  unspecified bind(F&& f, BoundArgs&&... bound_args);
```

- 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 == sizeof...(bound\_args).
- Returns: A forwarding call wrapper g with a nested type result\_type defined as a synonym for R. The effect of g(u1, u2, ..., uM) shall be INVOKE(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]
- 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 Vi is T&;
  - if the value of is\_bind\_expression<TiD>::value is true, the argument is tid(std::forward<Uj>(uj)...) and its type Vi is result\_of<TiD cv (Uj...)>::type;

§ 20.8.9.1.2 551

```
— 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 &.

## 20.8.9.1.3 Placeholders

[func.bind.place]

```
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.

## 20.8.10 Function template mem\_fn

[func.memfn]

```
template < class R, class T>
  unspecified mem_fn(R T::* pm);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::* pm)(Args...));
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::* pm)(Args...) const);
template < class R, class T, class... Args>
 unspecified mem_fn(R (T::* pm)(Args...) volatile);
template < class R, class T, class... Args>
 unspecified mem_fn(R (T::* pm)(Args...) const volatile);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::* pm)(Args...) &);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::* pm)(Args...) const &);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::* pm)(Args...) volatile &);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::* pm)(Args...) const volatile &);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::* pm)(Args...) &&);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::* pm)(Args...) const &&);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::* pm)(Args...) volatile &&);
template < class R, class T, class... Args>
  unspecified mem_fn(R (T::* pm)(Args...) const volatile &&);
     Returns: A simple call wrapper (20.8.1) fn such that the expression fn(t, a2, ..., aN) is equivalent
     to INVOKE (pm, t, a2, ..., aN) (20.8.2). fn shall have a nested type result_type that is a synonym
     for the return type of pm when pm is a pointer to member function.
```

§ 20.8.10 552

The simple call wrapper shall define two nested types named argument\_type and result\_type as synonyms for cv T\* and Ret, respectively, when pm is a pointer to member function with cv-qualifier cv and taking no arguments, where Ret is pm's return type.

- The simple call wrapper shall define three nested types named first\_argument\_type, second\_-argument\_type, and result\_type as synonyms for cv T\*, T1, and Ret, respectively, when pm is a pointer to member function with cv-qualifier cv and taking one argument of type T1, where Ret is pm's return type.
- 4 Throws: Nothing.

1

# 20.8.11 Polymorphic function wrappers

[func.wrap]

This subclause describes a polymorphic wrapper class that encapsulates arbitrary callable objects.

## 20.8.11.1 Class bad\_function\_call

[func.wrap.badcall]

An exception of type bad\_function\_call is thrown by function::operator() (20.8.11.2.4) when the function wrapper object has no target.

#### 20.8.11.2 Class template function

Effects: constructs a bad function call object.

[func.wrap.func]

```
namespace std {
  template<class> class function; // undefined
  template < class R, class... ArgTypes>
  class function<R(ArgTypes...)> {
  public:
    typedef R result_type;
                                          // iff sizeof...(ArgTypes) == 1 and
    typedef T1 argument_type;
                                          // the type in ArgTypes is T1
                                          // iff sizeof...(ArgTypes) == 2 and
    typedef T1 first_argument_type;
                                          // ArgTypes contains T1 and T2
                                         // iff sizeof...(ArgTypes) == 2 and
    typedef T2 second_argument_type;
                                          // ArgTypes contains T1 and T2
    // 20.8.11.2.1, construct/copy/destroy:
    function() noexcept;
    function(nullptr_t) noexcept;
    function(const function&);
    function(function&&);
    template<class F> function(F);
    template<class A> function(allocator_arg_t, const A&) noexcept;
    template<class A> function(allocator_arg_t, const A&,
      nullptr_t) noexcept;
```

§ 20.8.11.2 553

```
template < class A > function(allocator_arg_t, const A&,
    const function&);
  template < class A > function(allocator_arg_t, const A&,
    function&&);
  template < class F, class A > function(allocator_arg_t, const A&, F);
  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>) noexcept;
  ~function();
  // 20.8.11.2.2, function modifiers:
  void swap(function&) noexcept;
  template<class F, class A> void assign(F&&, const A&);
  // 20.8.11.2.3, function capacity:
  explicit operator bool() const noexcept;
  // 20.8.11.2.4, function invocation:
  R operator()(ArgTypes...) const;
  // 20.8.11.2.5, function target access:
  const std::type_info& target_type() const noexcept;
  template <typename T> const T* target() const noexcept;
};
// 20.8.11.2.6, Null pointer comparisons:
template <class R, class... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
template <class R, class... ArgTypes>
  bool operator==(nullptr_t, const function<R(ArgTypes...)>&) noexcept;
template <class R, class... ArgTypes>
  bool operator!=(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
template <class R, class... ArgTypes>
  bool operator!=(nullptr_t, const function<R(ArgTypes...)>&) noexcept;
// 20.8.11.2.7, 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 { };
```

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

}

§ 20.8.11.2 554

functions to be first-class objects.

<sup>2</sup> A callable object f of type F is *Callable* for argument types ArgTypes and return type R if the expression *INVOKE* (f, declval<ArgTypes>()..., R), considered as an unevaluated operand (Clause 5), is well formed (20.8.2).

- The function class template is a call wrapper (20.8.1) whose call signature (20.8.1) is R(ArgTypes...).

  20.8.11.2.1 function construct/copy/destroy [func.wrap.func.con]
- 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 17.6.3.5). A copy of the allocator argument is used to allocate memory, if necessary, for the internal data structures of the constructed function object.

```
function() noexcept;
   template <class A> function(allocator_arg_t, const A& a) noexcept;
        Postconditions: !*this.
   function(nullptr_t) noexcept;
   template <class A> function(allocator_arg_t, const A& a, nullptr_t) noexcept;
3
         Postconditions: !*this.
   function(const function& f);
   template <class A> function(allocator_arg_t, const A& a, const function& f);
4
        Postconditions: !*this if !f; otherwise, *this targets a copy of f.target().
5
         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(allocator_arg_t, const A& a, function&& f);
6
        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(allocator_arg_t, const A& a, F f);
        Requires: F shall be CopyConstructible. f shall be Callable (20.8.11.2) for argument types ArgTypes
        and return type R. The copy constructor and destructor of A shall not throw exceptions.
8
        Postconditions: !*this if any of the following hold:
          — f is a NULL function pointer.
```

§ 20.8.11.2.1 555

— f is a NULL pointer to member.

— F is an instance of the function class template, and !f

Otherwise, \*this targets a copy of f initialized with std::move(f). [Note: Implementations are

9

```
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]
10
         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);
11
         Effects: function(f).swap(*this);
12
         Returns: *this
   function& operator=(function&& f);
13
         Effects: Replaces the target of *this with the target of f.
         Returns: *this
   function& operator=(nullptr_t);
15
         Effects: If *this != NULL, destroys the target of this.
16
         Postconditions: !(*this).
17
         Returns: *this
   template<class F> function& operator=(F&& f);
18
         Effects: function(std::forward<F>(f)).swap(*this);
19
         Returns: *this
   template<class F> function& operator=(reference_wrapper<F> f) noexcept;
20
         Effects: function(f).swap(*this);
21
         Returns: *this
   ~function();
22
         Effects: If *this != NULL, destroys the target of this.
   20.8.11.2.2 function modifiers
                                                                                    [func.wrap.func.mod]
   void swap(function& other) noexcept;
1
         Effects: interchanges the targets of *this and other.
   template < class F, class A>
      void assign(F&& f, const A& a);
         Effects: function(allocator_arg, a, std::forward<F>(f)).swap(*this)
   § 20.8.11.2.2
                                                                                                        556
```

```
20.8.11.2.3 function capacity
                                                                                  [func.wrap.func.cap]
   explicit operator bool() const noexcept;
        Returns: true if *this has a target, otherwise false.
   20.8.11.2.4 function invocation
                                                                                  [func.wrap.func.inv]
   R operator()(ArgTypes... args) const
1
        Effects: INVOKE(f, std::forward<ArgTypes>(args)..., R) (20.8.2), where f is the target ob-
        ject (20.8.1) of *this.
2
        Returns: Nothing if R is void, otherwise the return value of INVOKE (f, std::forward<ArgTypes>(
        args)..., R).
3
        Throws: bad function call if !*this; otherwise, any exception thrown by the wrapped callable
        object.
   20.8.11.2.5 function target access
                                                                                 [func.wrap.func.targ]
   const std::type_info& target_type() const noexcept;
        Returns: If *this has a target of type T, typeid(T); otherwise, typeid(void).
   template<typename T>
                              T* target() noexcept;
   template<typename T> const T* target() const noexcept;
2
        Requires: T shall be a type that is Callable (20.8.11.2) for parameter types ArgTypes and return type
3
        Returns: If target type() == typeid(T) a pointer to the stored function target; otherwise a null
        pointer.
   20.8.11.2.6 null pointer comparison operators
                                                                              [func.wrap.func.nullptr]
   template <class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;
   template <class R, class... ArgTypes>
    bool operator==(nullptr_t, const function<R(ArgTypes...)>& f) noexcept;
        Returns: !f.
   template <class R, class... ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;
   template <class R, class... ArgTypes>
    bool operator!=(nullptr_t, const function<R(ArgTypes...)>& f) noexcept;
        Returns:
                   (bool) f.
                                                                                  [func.wrap.func.alg]
   20.8.11.2.7 specialized algorithms
   template < class R, class... ArgTypes>
     void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2);
        Effects: f1.swap(f2);
```

§ 20.8.11.2.7 557

## 20.8.12 Class template hash

[unord.hash]

The unordered associative containers defined in 23.5 use specializations of the class template hash as the default hash function. For all object types Key for which there exists a specialization hash<Key>, the instantiation hash<Key> shall:

- satisfy the Hash requirements (17.6.3.4), with Key as the function call argument type, the Default-Constructible requirements (Table 19), the CopyAssignable requirements (Table 23),
- be swappable (17.6.3.2) 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;
- satisfy the requirement that the expression h(k), where h is an object of type hash<Key> and k is an object of type Key, shall not throw an exception unless hash<Key> is a user-defined specialization that depends on at least one user-defined type.

```
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 <class T> struct hash<T*>;
```

Requires: the template specializations shall meet the requirements of class template hash (20.8.12).

## 20.9 Metaprogramming and type traits

[meta]

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.

## 20.9.1 Requirements

[meta.rqmts]

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

§ 20.9.1 558

shall be DefaultConstructible, CopyConstructible, and publicly and unambiguously derived, directly or indirectly, from its BaseCharacteristic, which is a specialization of the template integral\_constant (20.9.3), with the arguments to the template integral\_constant determined by the requirements for the particular property being described. The member names of the BaseCharacteristic shall not be hidden and shall be unambiguously available in the UnaryTypeTrait.

- A BinaryTypeTrait describes a relationship between two types. It shall be a class template that takes two template type arguments and, optionally, additional arguments that help define the relationship being described. It shall be DefaultConstructible, CopyConstructible, and publicly and unambiguously derived, directly or indirectly, from its BaseCharacteristic, which is a specialization of the template integral\_constant (20.9.3), with the arguments to the template integral\_constant determined by the requirements for the particular relationship being described. The member names of the BaseCharacteristic shall not be hidden and shall be unambiguously available in the BinaryTypeTrait.
- A Transformation Trait 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.

## 20.9.2 Header <type\_traits> synopsis

[meta.type.synop]

```
namespace std {
  // 20.9.3, helper class:
  template <class T, T v> struct integral_constant;
 typedef integral_constant<bool, true> true_type;
  typedef integral_constant<bool, false> false_type;
  // 20.9.4.1, 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;
  // 20.9.4.2, 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;
  // 20.9.4.3, 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;
```

§ 20.9.2 559

```
template <class T> struct is_literal_type;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T, class... Args> struct is_constructible;
template <class T> struct is_default_constructible;
template <class T> struct is_copy_constructible;
template <class T> struct is_move_constructible;
template <class T, class U> struct is_assignable;
template <class T> struct is_copy_assignable;
template <class T> struct is_move_assignable;
template <class T> struct is_destructible;
template <class T, class... Args> struct is_trivially_constructible;
template <class T> struct is_trivially_default_constructible;
template <class T> struct is_trivially_copy_constructible;
template <class T> struct is_trivially_move_constructible;
template <class T, class U> struct is_trivially_assignable;
template <class T> struct is_trivially_copy_assignable;
template <class T> struct is_trivially_move_assignable;
template <class T> struct is_trivially_destructible;
template <class T, class... Args> struct is_nothrow_constructible;
template <class T> struct is_nothrow_default_constructible;
template <class T> struct is_nothrow_copy_constructible;
template <class T> struct is_nothrow_move_constructible;
template <class T, class U> struct is_nothrow_assignable;
template <class T> struct is_nothrow_copy_assignable;
template <class T> struct is_nothrow_move_assignable;
template <class T> struct is_nothrow_destructible;
template <class T> struct has_virtual_destructor;
// 20.9.5, type property queries:
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;
// 20.9.6, type relations:
template <class T, class U> struct is_same;
template <class Base, class Derived> struct is_base_of;
template <class From, class To> struct is_convertible;
// 20.9.7.1, const-volatile modifications:
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
```

§ 20.9.2 560

```
template <class T> struct add_const;
 template <class T> struct add_volatile;
 template <class T> struct add_cv;
 // 20.9.7.2, reference modifications:
 template <class T> struct remove_reference;
 template <class T> struct add_lvalue_reference;
 template <class T> struct add_rvalue_reference;
 // 20.9.7.3, sign modifications:
 template <class T> struct make_signed;
 template <class T> struct make_unsigned;
  // 20.9.7.4, array modifications:
 template <class T> struct remove_extent;
 template <class T> struct remove_all_extents;
 // 20.9.7.5, pointer modifications:
 template <class T> struct remove_pointer;
 template <class T> struct add_pointer;
  // 20.9.7.6, other transformations:
 template <std::size_t Len,
            std::size_t Align = default-alignment>
                                                     // see 20.9.7.6
   struct aligned_storage;
 template <std::size_t Len, class... Types> struct aligned_union;
 template <class T> struct decay;
 template <bool, class T = void> struct enable_if;
 template <bool, class T, class F> struct conditional;
 template <class... T> struct common_type;
 template <class T> struct underlying_type;
 template <class> class result_of; // not defined
 template <class F, class... ArgTypes> class result_of<F(ArgTypes...)>;
} // namespace std
```

<sup>1</sup> The behavior of a program that adds specializations for any of the class templates defined in this subclause is undefined unless otherwise specified.

## 20.9.3 Helper classes

[meta.help]

```
namespace std {
  template <class T, T v>
  struct integral_constant {
    static constexpr T value = v;
    typedef T value_type;
    typedef integral_constant<T,v> type;
    constexpr operator value_type() { 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.

## 20.9.4 Unary type traits

[meta.unary]

<sup>1</sup> This sub-clause contains templates that may be used to query the properties of a type at compile time.

§ 20.9.4 561

<sup>2</sup> Each of these templates shall be a UnaryTypeTrait (20.9.1) with a BaseCharacteristic of true\_type if the corresponding condition is true, otherwise false\_type.

## 20.9.4.1 Primary type categories

[meta.unary.cat]

- 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 47 — Primary type category predicates

Template	Condition	Comments
template <class t=""></class>	T is void	
struct is_void;		
template <class t=""></class>	T is an integral type (3.9.1)	
struct is_integral;		
template <class t=""></class>	T is a floating point	
<pre>struct is_floating_point;</pre>	type $(3.9.1)$	
template <class t=""></class>	T is an array type (3.9.2) of	Class template
struct is_array;	known or unknown extent	array $(23.3.2)$ is not an array type.
template <class t=""></class>	T is a pointer type (3.9.2)	Includes pointers to
struct is_pointer;		functions but not pointers
		to non-static members.
template <class t=""></class>	T is an lvalue reference	
struct is_lvalue_reference;	type $(8.3.2)$	
template <class t=""></class>	T is an rvalue reference	
struct is_rvalue_reference;	type $(8.3.2)$	
template <class t=""></class>	T is a pointer to non-static	
<pre>struct is_member_object_pointer;</pre>	data member	
template <class t=""></class>	T is a pointer to non-static	
struct	member function	
<pre>is_member_function_pointer;</pre>		
template <class t=""></class>	T is an enumeration	
struct is_enum;	type $(3.9.2)$	
template <class t=""></class>	T is a union type $(3.9.2)$	
struct is_union;		
template <class t=""></class>	T is a class type but not a	
struct is_class;	union type $(3.9.2)$	
template <class t=""></class>	T is a function type $(3.9.2)$	
struct is_function;		

## 20.9.4.2 Composite type traits

[meta.unary.comp]

- <sup>1</sup> These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section 3.9.
- <sup>2</sup> 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.

§ 20.9.4.2 562

Table 48 —	Composite	type	category	predicates

Template	Condition	Comments
template <class t=""></class>	T is an lvalue reference or	
struct is_reference;	an rvalue reference	
template <class t=""></class>	T is an arithmetic	
struct is_arithmetic;	type $(3.9.1)$	
template <class t=""></class>	T is a fundamental	
<pre>struct is_fundamental;</pre>	type $(3.9.1)$	
template <class t=""></class>	T is an object type (3.9)	
struct is_object;		
template <class t=""></class>	T is a scalar type (3.9)	
struct is_scalar;		
template <class t=""></class>	T is a compound	
struct is_compound;	type $(3.9.2)$	
template <class t=""></class>	T is a pointer to non-static	
struct is_member_pointer;	data member or non-static	
	member function	

## 20.9.4.3 Type properties

[meta.unary.prop]

- These templates provide access to some of the more important properties of types.
- <sup>2</sup> It is unspecified whether the library defines any full or partial specializations of any of these templates.
- For all of the class templates X declared in this Clause, instantiating that template with a template-argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of X require that the argument must be a complete type.

Table 49 — Type property predicates

Template	Condition	Preconditions
template <class t=""></class>	T is const-qualified (3.9.3)	
struct is_const;		
template <class t=""></class>	T is	
struct is_volatile;	volatile-qualified $(3.9.3)$	
template <class t=""></class>	T is a trivial type $(3.9)$	T shall be a complete type,
struct is_trivial;		(possibly cv-qualified)
		void, or an array of
		unknown bound.
template <class t=""></class>	T is a trivially copyable	T shall be a complete type,
struct is_trivially_copyable;	type $(3.9)$	(possibly cv-qualified)
		void, or an array of
		unknown bound.
template <class t=""></class>	T is a standard-layout	T shall be a complete type,
struct is_standard_layout;	type $(3.9)$	(possibly cv-qualified)
		void, or an array of
		unknown bound.
template <class t=""></class>	T is a POD type (3.9)	T shall be a complete type,
struct is_pod;		(possibly cv-qualified)
		void, or an array of
		unknown bound.

 $\odot$  ISO/IEC N3337

Table 49 — Type property predicates (continued)

Template	Condition	Preconditions
<pre>template <class t=""> struct is_literal_type;</class></pre>	T is a literal type (3.9)	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_empty;</class></pre>	T is a class type, but not a union type, with no non-static data members other than bit-fields of length 0, no virtual member functions, no virtual base classes, and no base class B for which is_empty <b>::value is false.</b>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_polymorphic;</class></pre>	T is a polymorphic class (10.3)	T shall be a complete type, type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_abstract;</class></pre>	T is an abstract class (10.4)	T shall be a complete type, type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_signed;</class></pre>	<pre>is arithmetic<t>::value &amp;&amp; T(-1) &lt; T(0)</t></pre>	
<pre>template <class t=""> struct is_unsigned;</class></pre>	<pre>is arithmetic<t>::value &amp;&amp; T(0) &lt; T(-1)</t></pre>	
<pre>template <class args="" class="" t,=""> struct is_constructible;</class></pre>	see below	T and all types in the parameter pack Args shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.
<pre>template <class t=""> struct is_default_constructible;</class></pre>	<pre>is constructible<t>::value is true.</t></pre>	T shall be a complete type, (possibly <i>cv</i> -qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_copy_constructible;</class></pre>	is_constructible <t, const T&amp;&gt;::value is true.</t, 	T shall be a complete type, (possibly <i>cv</i> -qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_move_constructible;</class></pre>	is_constructible <t, T&amp;&amp;&gt;::value is true.</t, 	T shall be a complete type, (possibly <i>cv</i> -qualified) void, or an array of unknown bound.

Table 49 — Type property predicates (continued)

Template	Condition	Preconditions
Template  template <class class="" t,="" u=""> struct is_assignable;</class>	The expression declval <t>() = declval<u>() is well-formed when treated as an unevaluated operand (Clause 5). Access checking is performed as if in a context unrelated to T and U. Only the validity of the immediate context of the assignment expression is considered. [Note: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and</u></t>	Preconditions  T and U shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.
<pre>template <class t=""> struct is_copy_assignable;  template <class t=""></class></class></pre>	can result in the program being ill-formed. — end note]  is_assignable <t&, const="" t&="">::value is true.  is_assignable<t&,< th=""><th>T shall be a complete type, (possibly <i>cv</i>-qualified) void, or an array of unknown bound.  T shall be a complete type,</th></t&,<></t&,>	T shall be a complete type, (possibly <i>cv</i> -qualified) void, or an array of unknown bound.  T shall be a complete type,
struct is_move_assignable;	T&&>::value is true.	(possibly <i>cv</i> -qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_destructible;</class></pre>	For a complete type T and given  template <class u="">  struct test { U u; };,  test<t>::~test() is not deleted.</t></class>	T shall be a complete type, (possibly <i>cv</i> -qualified) void, or an array of unknown bound.

 $\odot$  ISO/IEC N3337

Table 49 — Type property predicates (continued)

Template	Condition	Preconditions
<pre>template <class args="" class="" t,=""> struct is_trivially_constructible;</class></pre>	is_constructible <t, args="">::value is true and the variable definition for is_constructible, as defined below, is known to call no operation that is not trivial (3.9, 12).</t,>	T and all types in the parameter pack Args shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.
<pre>template <class t=""> struct is_trivially_default_constructible;</class></pre>	<pre>is_trivially constructible<t>::value is true.</t></pre>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_trivially_copy_constructible;</class></pre>	is_trivially constructible <t, const="" t&="">::value is true.</t,>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_trivially_move_constructible;</class></pre>	is_trivially constructible <t, t&&="">::value is true.</t,>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class class="" t,="" u=""> struct is_trivially_assignable;</class></pre>	is_assignable <t, U&gt;::value is true and the assignment, as defined by is_assignable, is known to call no operation that is not trivial (3.9, 12).</t, 	T and U shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.
<pre>template <class t=""> struct is_trivially_copy_assignable;</class></pre>	is_trivially assignable <t&, const="" t&="">::value is true.</t&,>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_trivially_move_assignable;</class></pre>	is_trivially assignable <t&, t&&="">::value is true.</t&,>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_trivially_destructible;</class></pre>	is destructible <t>::value is true and the indicated destructor is known to be trivial.</t>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class args="" class="" t,=""> struct is_nothrow_constructible;</class></pre>	is_constructible <t, args="">::value is true and the variable definition for is_constructible, as defined below, is known not to throw any exceptions (5.3.7).</t,>	T and all types in the parameter pack Args shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.

Table 49 — Type property predicates (continued)

Template	Condition	Preconditions
<pre>template <class t=""> struct is_nothrow_default_constructible;</class></pre>	<pre>is_nothrow constructible<t>::value is true.</t></pre>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_nothrow_copy_constructible;</class></pre>	is_nothrow constructible <t, const="" t&="">::value is true.</t,>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_nothrow_move_constructible;</class></pre>	is_nothrow constructible <t, t&&="">::value is true.</t,>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class class="" t,="" u=""> struct is_nothrow_assignable;</class></pre>	is_assignable <t, U&gt;::value is true and the assignment is known not to throw any exceptions (5.3.7).</t, 	T and U shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.
<pre>template <class t=""> struct is_nothrow_copy_assignable;</class></pre>	is_nothrow assignable <t&, const="" t&="">::value is true.</t&,>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_nothrow_move_assignable;</class></pre>	is_nothrow assignable <t&, t&&="">::value is true.</t&,>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct is_nothrow_destructible;</class></pre>	is destructible <t>::value is true and the indicated destructor is known not to throw any exceptions (5.3.7).</t>	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.
<pre>template <class t=""> struct has_virtual_destructor;</class></pre>	T has a virtual destructor (12.4)	T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.

## 4 [Example:

```
is_const<const volatile int>::value
                                           //\ true
 is_const<const int*>::value
                                           // false
 is_const<const int&>::value
                                           // false
                                           // false
// true
 is_const<int[3]>::value
 is_const<const int[3]>::value
— end example]
```

# <sup>5</sup> [Example:

```
{\tt remove\_const} {\tt <const} \ {\tt volatile} \ {\tt int} {\tt >::type} \ \ /\!/ \ {\tt volatile} \ {\tt int}
remove_const<const int* const>::type // const int*
```

the predicate condition for a template specialization is\_constructible<T, Args...> shall be satisfied if and only if the following variable definition would be well-formed for some invented variable t:

```
T t(create<Args>()...);
```

[Note: These tokens are never interpreted as a function declaration. —  $end\ note$ ] Access checking is performed as if in a context unrelated to T and any of the Args. Only the validity of the immediate context of the variable initialization is considered. [Note: The evaluation of the initialization can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. —  $end\ note$ ]

## 20.9.5 Type property queries

[meta.unary.prop.query]

This sub-clause contains templates that may be used to query properties of types at compile time.

Table 50 — Type property queries

Template	Value
template <class t=""></class>	alignof(T).
<pre>struct alignment_of;</pre>	Requires:alignof(T) shall be a valid expression (5.3.6)
template <class t=""></class>	If T names an array type, an integer value representing the number of
struct rank;	dimensions of T; otherwise, 0.
template <class t,<="" td=""><td>If T is not an array type, or if it has rank less than or equal to I, or if I</td></class>	If T is not an array type, or if it has rank less than or equal to I, or if I
unsigned I = 0>	is 0 and T has type "array of unknown bound of U", then 0; otherwise,
struct extent;	the bound (8.3.4) of the I'th dimension of T, where indexing of I is
	zero-based

<sup>2</sup> [Example:

```
// the following assertions hold:
assert(rank<int>::value == 0);
assert(rank<int[2]>::value == 1);
assert(rank<int[] [4]>::value == 2);

-- end example]

[Example:
    // the following assertions hold:
assert(extent<int>::value == 0);
assert(extent<int[2]>::value == 2);
assert(extent<int[2] [4]>::value == 2);
assert(extent<int[] [4]>::value == 0);
assert((extent<int, 1>::value) == 0);
assert((extent<int[2], 1>::value) == 0);
assert((extent<int[2], 1>::value) == 4);
assert((extent<int[] [4], 1>::value) == 4);
```

§ 20.9.5

— end example]

### 20.9.6 Relationships between types

[meta.rel]

This sub-clause contains templates that may be used to query relationships between types at compile time.

<sup>2</sup> Each of these templates shall be a BinaryTypeTrait (20.9.1) with a BaseCharacteristic of true\_type if the corresponding condition is true, otherwise false\_type.

Table 51 — Type relationship predicates

Template	Condition	Comments
<pre>template <class class="" t,="" u=""> struct is_same;</class></pre>	T and U name the same type with the same cv-qualifications	
<pre>template <class base,="" class="" derived=""> struct is_base_of;</class></pre>	Base is a base class of Derived (10) without regard to cv-qualifiers or Base and Derived are not unions and name the same class type without regard to cv-qualifiers	If Base and Derived are class types and are different types (ignoring possible cv-qualifiers) then Derived shall be a complete type. [Note: Base classes that are private, protected, or ambiguous are, nonetheless, base classes. — end note]
<pre>template <class class="" from,="" to=""> struct is_convertible;</class></pre>	see below	From and To shall be complete types, arrays of unknown bound, or (possibly cv-qualified) void types.

3 [Example:

```
struct B {};
 struct B1 : B {};
 struct B2 : B {};
 struct D : private B1, private B2 {};
 is_base_of<B, D>::value
                                  // true
                                   // true
 is_base_of<const B, D>::value
                                  // true
 is_base_of<B, const D>::value
 is_base_of<B, const B>::value
                                  // true
 is_base_of<D, B>::value
                                  // false
                                  // false
 is_base_of<B&, D&>::value
                                  // false
 is_base_of<B[3], D[3]>::value
 is_base_of<int, int>::value
                                  // false
— end example]
```

<sup>4</sup> Given the following function prototype:

```
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:

```
To test() {
   return create<From>();
}
```

§ 20.9.6 569

[Note: This requirement gives well defined results for reference types, void types, array types, and function types. —  $end\ note$ ] Access checking is performed as if in a context unrelated to To and From. Only the validity of the immediate context of the expression of the return-statement (including conversions to the return type) is considered. [Note: The evaluation of the conversion can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. —  $end\ note$ ]

### 20.9.7 Transformations between types

[meta.trans]

- <sup>1</sup> This sub-clause contains templates that may be used to transform one type to another following some predefined rule.
- <sup>2</sup> Each of the templates in this subclause shall be a *TransformationTrait* (20.9.1).

### 20.9.7.1 Const-volatile modifications

[meta.trans.cv]

Table 52 — Const-volatile modifications

Template	Comments
template <class t=""></class>	The member typedef type shall name the same type as T except that
struct remove_const;	any top-level const-qualifier has been removed.
	[ Example:remove_const <const int="" volatile="">::type evaluates to</const>
	volatile int, whereas remove_const <const int*="">::type evaluates</const>
	to const int*. — end example]
template <class t=""></class>	The member typedef type shall name the same type as T except that
struct remove_volatile;	any top-level volatile-qualifier has been removed.
	[ Example:remove_volatile < const volatile int>::type evaluates to
	<pre>const int, whereas remove_volatile <volatile int*="">::type</volatile></pre>
	evaluates to volatile int*. — end example]
template <class t=""></class>	The member typedef type shall be the same as T except that any
struct remove_cv;	top-level cv-qualifier has been removed. [Example:remove_cv <const< th=""></const<>
	volatile int>::type evaluates to int, whereas remove_cv <const< th=""></const<>
	volatile int*>::type evaluates to const volatile int*. $-end$
	example]
template <class t=""></class>	If T is a reference, function, or top-level const-qualified type, then type
struct add_const;	shall name the same type as T, otherwise T const.
template <class t=""></class>	If T is a reference, function, or top-level volatile-qualified type, then
struct add_volatile;	type shall name the same type as T, otherwise T volatile.
template <class t=""></class>	The member typedef type shall name the same type as
struct add_cv;	add_const <typename add_volatile<t="">::type&gt;::type.</typename>

#### 20.9.7.2 Reference modifications

[meta.trans.ref]

Table 53 — Reference modifications

Template	Comments
template <class t=""></class>	If T has type "reference to T1" then the member typedef type shall
struct remove_reference;	name T1; otherwise, type shall name T.
template <class t=""></class>	If T names an object or function type then the member typedef type
struct	shall name T&; otherwise, if T names a type "rvalue reference to T1" then
add_lvalue_reference;	the member typedef type shall name T1&; otherwise, type shall name T.

§ 20.9.7.2 570

 $\odot$  ISO/IEC N3337

Table 53 — Reference modifications (continued)

Template	Comments
template <class t=""></class>	If T names an object or function type then the member typedef type shall name T&&; otherwise, type shall name T. [Note: This rule reflects
add_rvalue_reference;	the semantics of reference collapsing (8.3.2). For example, when a type T
	names a type T1&, the type add_rvalue_reference <t>::type is not an rvalue reference. — end note]</t>

# 20.9.7.3 Sign modifications

 $[{
m meta.trans.sign}]$ 

Table 54 — Sign modifications

Template	Comments		
template <class t=""></class>	If T names a (possibly cv-qualified) signed integer type (3.9.1) then the		
struct make_signed;	member typedef type shall name the type T; otherwise, if T names a		
	(possibly cv-qualified) unsigned integer type then type shall name the		
	corresponding signed integer type, with the same cv-qualifiers as T;		
	otherwise, type shall name the signed integer type with smallest		
	rank (4.13) for which sizeof(T) == sizeof(type), with the same		
	cv-qualifiers as T.		
	Requires: T shall be a (possibly cv-qualified) integral type or enumeration		
	but not a bool type.		
template <class t=""></class>	If T names a (possibly cv-qualified) unsigned integer type (3.9.1) then		
struct make_unsigned;	the member typedef type shall name the type T; otherwise, if T names a		
	(possibly cv-qualified) signed integer type then type shall name the		
	corresponding unsigned integer type, with the same cv-qualifiers as T;		
	otherwise, type shall name the unsigned integer type with smallest		
	rank (4.13) for which sizeof(T) == sizeof(type), with the same		
	cv-qualifiers as T.		
	Requires: T shall be a (possibly cv-qualified) integral type or enumeration		
	but not a bool type.		

§ 20.9.7.3 571

### 20.9.7.4 Array modifications

[meta.trans.arr]

Table 55 — Array modifications

Template	Comments	
template <class t=""></class>	If T names a type "array of U", the member typedef type shall be U,	
struct remove_extent;	otherwise T. [Note: For multidimensional arrays, only the first array	
	dimension is removed. For a type "array of const U", the resulting type	
	is const U. — end note]	
template <class t=""></class>	If T is "multi-dimensional array of U", the resulting member typedef	
struct remove_all_extents;	type is U, otherwise T.	

## [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]
```

### 20.9.7.5 Pointer modifications

[meta.trans.ptr]

Table 56 — Pointer modifications

Template	Comments	
template <class t=""></class>	If T has type "(possibly cv-qualified) pointer to T1" then the member	
struct remove_pointer;	typedef type shall name T1; otherwise, it shall name T.	
template <class t=""></class>	The member typedef type shall name the same type as	
struct add_pointer;	remove_reference <t>::type*.</t>	

§ 20.9.7.5 572

 $\odot$  ISO/IEC N3337

# 20.9.7.6 Other transformations

[meta.trans.other]

Table 57 — Other transformations

Template	Condition	Comments
<pre>template <std::size_t align="default-alignment" len,="" std::size_t=""> struct aligned_storage;</std::size_t></pre>	Len shall not be zero. Align shall be equal to alignof(T) for some type T or to default-alignment.	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.
<pre>template <std::size_t class="" len,="" types=""> struct aligned_union;</std::size_t></pre>	At least one type is provided.	The member typedef type shall be a POD type suitable for use as uninitialized storage for any object whose type is listed in Types; its size shall be at least Len. The static member alignment_value shall be an integral constant of type std::size_t whose value is the strictest alignment of all types listed in Types.
template <class t=""> struct</class>		Let U be
decay;		remove_reference <t>::type. If is_array<u>::value is true, the member typedef type shall equal remove_extent<u>::type*. If is_function<u>::value is true, the member typedef type shall equal add_pointer<u>::type. Otherwise the member typedef type equals remove_cv<u>::type. [Note: This behavior is similar to the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) conversions applied when an lvalue expression is used as an rvalue, but also strips cv-qualifiers from class types in order to more closely model by-value argument passing. — end note]</u></u></u></u></u></t>
template <bool b,="" class="" t="&lt;/td"><td></td><td>If B is true, the member typedef type</td></bool>		If B is true, the member typedef type
void> struct enable_if;		shall equal T; otherwise, there shall be no member typedef type.
template <bool b,="" class="" t,<="" td=""><td></td><td>If B is true, the member typedef type</td></bool>		If B is true, the member typedef type
class F> struct conditional;		shall equal T. If B is false, the member typedef type shall equal F.

§ 20.9.7.6 573

Table 57 — Other transformations (continued)

Template	Condition	Comments	
template <class t=""></class>		The member typedef type shall be	
<pre>struct common_type;</pre>		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. $-end note$ ]	
template <class t=""></class>	T shall be an enumeration	The member typedef type shall name	
struct underlying_type;	type $(7.2)$	the underlying type of T.	
template <class fn,<="" td=""><td>Fn shall be a callable</td><td>The member typedef type shall name</td></class>	Fn shall be a callable	The member typedef type shall name	
class ArgTypes> struct	type $(20.8.1)$ , reference to	the type	
<pre>result_of<fn(argtypes)>;</fn(argtypes)></pre>	function, or reference to	<pre>decltype(INVOKE(declval<fn>(),</fn></pre>	
	callable type. The	<pre>declval<argtypes>()))</argtypes></pre>	
	expression		
	<pre>decltype(INVOKE(declval<fn>(),</fn></pre>		
	<pre>declval<argtypes>()))</argtypes></pre>		
	shall be well formed.		

[Note: A typical implementation would define aligned\_storage as:

```
template <std::size_t Len, std::size_t Alignment>
     struct aligned_storage {
       typedef struct {
         alignas(Alignment) unsigned char __data[Len];
       } type;
    };
   - end note]
<sup>2</sup> It is implementation-defined whether any extended alignment is supported (3.11).
```

- <sup>3</sup> The nested typedef common\_type::type shall be defined as follows:

```
template <class ...T> struct common_type;
template <class T>
struct common_type<T> {
 typedef T type;
};
template <class T, class U>
struct common_type<T, U> {
 typedef decltype(true ? declval<T>() : declval<U>()) type;
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;
};
```

§ 20.9.7.6 574

```
typedef bool (&PF1)();
typedef short (*PF2)(long);

struct S {
   operator PF2() const;
   double operator()(char, int&);
   void fn(long) const;
   char data;
};

typedef void (S::*PMF)(long) const;
typedef char S::*PMD;

the following assertions will hold:
```

static\_assert(is\_same<result\_of<S&(unsigned char, int&)>::type, double>::value, "Error!");

static\_assert(is\_same<result\_of<PMF(unique\_ptr<S>, int)>::type, void>::value, "Error!");

static\_assert(is\_same<result\_of<PMD(const S\*)>::type, const char&>::value, "Error!");

static\_assert(is\_same<result\_of<S(int)>::type, short>::value, "Error!");

static\_assert(is\_same<result\_of<PMD(S)>::type, char&&>::value, "Error!");

static\_assert(is\_same<result\_of<PF1()>::type, bool>::value, "Error!");

— end example]

### 20.10 Compile-time rational arithmetic

[ratio]

### 20.10.1 In general

[Example: Given these definitions:

[ratio.general]

- 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.
- <sup>2</sup> Throughout this subclause, if the template argument types R1 and R2 are not specializations of the ratio template, the program is ill-formed.

### 20.10.2 Header <ratio> synopsis

[ratio.syn]

```
namespace std {
  // 20.10.3, class template ratio
  template <intmax_t N, intmax_t D = 1> class ratio;
  // 20.10.4, ratio arithmetic
  template <class R1, class R2> using ratio_add = see below;
  template <class R1, class R2> using ratio_subtract = see below;
  template <class R1, class R2> using ratio_multiply = see below;
  template <class R1, class R2> using ratio_divide = see below;
  // 20.10.5, ratio comparison
  template <class R1, class R2> struct ratio_equal;
  template <class R1, class R2> struct ratio_not_equal;
  template <class R1, class R2> struct ratio_less;
  template <class R1, class R2> struct ratio_less_equal;
  template <class R1, class R2> struct ratio_greater;
  template <class R1, class R2> struct ratio_greater_equal;
  // 20.10.6, convenience SI typedefs
  typedef ratio<1, 1000000000000000000000000000000000 yocto; // see below
                       10000000000000000000000000000000> zepto; // see below
  typedef ratio<1,</pre>
```

§ 20.10.2 575

```
typedef ratio<1,
                      typedef ratio<1,
                         10000000000000000> femto:
typedef ratio<1,
                            1000000000000> pico;
typedef ratio<1,
                               1000000000> nano;
                                  1000000> micro;
typedef ratio<1,
typedef ratio<1,
                                     1000> milli;
typedef ratio<1,
                                      100> centi;
typedef ratio<1,
                                       10> deci;
typedef ratio<
                                    10, 1> deca;
                                   100, 1> hecto;
typedef ratio<
                                  1000, 1> kilo;
typedef ratio<
                               1000000, 1> mega;
typedef ratio<
                            1000000000, 1> giga;
typedef ratio<
                         1000000000000, 1> tera;
typedef ratio<
typedef ratio<
                      1000000000000000, 1> peta;
typedef ratio<
                   1000000000000000000, 1> exa;
                1000000000000000000000, 1> zetta; // see below
typedef ratio<
typedef ratio<10000000000000000000000, 1> yotta; // see below
```

### 20.10.3 Class template ratio

[ratio.ratio]

```
namespace std {
  template <intmax_t N, intmax_t D = 1>
  class ratio {
  public:
    typedef ratio<num, den> type;
    static constexpr intmax_t num;
    static constexpr intmax_t den;
  };
}
```

- 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]
- 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.

### 20.10.4 Arithmetic on ratios

[ratio.arithmetic]

- Each of the alias templates ratio\_add, ratio\_subtract, ratio\_multiply, and ratio\_divide denotes the result of an arithmetic computation on two ratios R1 and R2. With X and Y computed (in the absence of arithmetic overflow) as specified by Table 58, each alias denotes a ratio<U, V> such that U is the same as ratio<X, Y>::num and V is the same as ratio<X, Y>::den.
- If it is not possible to represent U or V with intmax\_t, the program is ill-formed. Otherwise, an implementation should yield correct values of U and V. If it is not possible to represent X or Y with intmax\_t, the program is ill-formed unless the implementation yields correct values of U and V.

<sup>3</sup> [Example:

```
static_assert(ratio_add<ratio<1,3>, ratio<1,6>>::num == 1, "1/3+1/6 == 1/2");
static_assert(ratio_add<ratio<1,3>, ratio<1,6>>::den == 2, "1/3+1/6 == 1/2");
static_assert(ratio_multiply<ratio<1,3>, ratio<3,2>>::num == 1, "1/3*3/2 == 1/2");
```

§ 20.10.4 576

(SO) = N3337

Table 58 — Expressions used to perform ratio a	arithmetic
--	------------

Type	Value of X	Value of Y
ratio_add <r1, r2=""></r1,>	R1::num * R2::den +	R1::den * R2::den
	R2::num * R1::den	
ratio_subtract <r1, r2=""></r1,>	R1::num * R2::den -	R1::den * R2::den
	R2::num * R1::den	
ratio_multiply <r1, r2=""></r1,>	R1::num * R2::num	R1::den * R2::den
ratio_divide <r1, r2=""></r1,>	R1::num * R2::den	R1::den * R2::num

```
static_assert(ratio_multiply<ratio<1,3>, ratio<3,2>>::den == 2, "1/3*3/2 == 1/2");
    // The following cases may cause the program to be ill-formed under some implementations
  static_assert(ratio_add<ratio<1,INT_MAX>, ratio<1,INT_MAX>>::num == 2,
   "1/MAX+1/MAX == 2/MAX");
  static_assert(ratio_add<ratio<1,INT_MAX>, ratio<1,INT_MAX>>::den == INT_MAX,
    "1/MAX+1/MAX == 2/MAX");
  static_assert(ratio_multiply<ratio<1,INT_MAX>, ratio<INT_MAX,2>>::num == 1,
    "1/MAX * MAX/2 == 1/2");
  static_assert(ratio_multiply<ratio<1,INT_MAX>, ratio<INT_MAX,2>>::den == 2,
    "1/MAX * MAX/2 == 1/2");
   — end example]
20.10.5 Comparison of ratios
                                                                               [ratio.comparison]
template <class R1, class R2> struct ratio_equal
  : integral_constant<bool, see below> { };
     If R1::num == R2::num and R1::den == R2::den, ratio_equal<R1, R2> shall be derived from
     integral_constant<br/>bool, true>; otherwise it shall be derived from integral_constant<br/>bool,
     false>.
template <class R1, class R2> struct ratio_not_equal
  : integral_constant<bool, !ratio_equal<R1, R2>::value> { };
template <class R1, class R2> struct ratio_less
  : integral_constant<bool, see below> { };
     If R1::num * R2::den < R2::num * R1::den, ratio_less<R1, R2> shall be derived from integral_-
     constant<bool, true>; otherwise it shall be derived from integral_constant<bool, false>. Im-
     plementations 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> { };
```

§ 20.10.5

### 20.10.6 SI types for ratio

[ratio.si]

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.

### 20.11 Time utilities

[time]

### **20.11.1** In general

[time.general]

<sup>1</sup> This subclause describes the chrono library (20.11.2) and various C functions (20.11.8) that provide generally useful time utilities.

### 20.11.2 Header <chrono> synopsis

[time.syn]

```
namespace std {
namespace chrono {
// 20.11.5, class template duration
template <class Rep, class Period = ratio<1> > class duration;
// 20.11.6, class template time_point
template <class Clock, class Duration = typename Clock::duration> class time_point;
} // namespace chrono
// 20.11.4.3 common_type specializations
template <class Rep1, class Period1, class Rep2, class Period2>
  struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>>;
template <class Clock, class Duration1, class Duration2>
  struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2>>;
namespace chrono {
// 20.11.4, customization traits
template <class Rep> struct treat_as_floating_point;
template <class Rep> struct duration_values;
// 20.11.5.5, duration arithmetic
template <class Rep1, class Period1, class Rep2, class Period2>
  typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
  constexpr operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
  constexpr operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period, class Rep2>
  duration<typename common_type<Rep1, Rep2>::type, Period>
  constexpr operator*(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Rep2, class Period>
  duration<typename common_type<Rep1, Rep2>::type, Period>
  constexpr operator*(const Rep1& s, const duration<Rep2, Period>& d);
template <class Rep1, class Period, class Rep2>
  duration<typename common_type<Rep1, Rep2>::type, Period>
  constexpr operator/(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Period1, class Rep2, class Period2>
  typename common_type<Rep1, Rep2>::type
  \verb|constexpr|| operator/(const|| duration < Rep1, Period1 > \& lhs, const|| duration < Rep2, Period2 > \& rhs); \\
template <class Rep1, class Period, class Rep2>
```

§ 20.11.2 578

```
duration<typename common_type<Rep1, Rep2>::type, Period>
  constexpr operator%(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Period1, class Rep2, class Period2>
 typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
 constexpr operator%(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
// 20.11.5.6, duration comparisons
template <class Rep1, class Period1, class Rep2, class Period2>
 constexpr bool operator==(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
 constexpr bool operator!=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
 constexpr bool operator< (const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
 constexpr bool operator <= (const duration < Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
 constexpr bool operator> (const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
 constexpr bool operator>=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
// 20.11.5.7, duration_cast
template <class ToDuration, class Rep, class Period>
 constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
// convenience typedefs
typedef duration<signed integer type of at least 64 bits,
                                                                nano> nanoseconds;
typedef duration<signed integer type of at least 55 bits,
                                                                micro> microseconds;
typedef duration<signed integer type of at least 45 bits,
                                                               milli> milliseconds;
typedef duration<signed integer type of at least 35 bits
                                                                       > seconds:
typedef duration<signed integer type of at least 29 bits, ratio< 60>> minutes;
typedef duration<signed integer type of at least 23 bits, ratio<3600>> hours;
// 20.11.6.5, 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);
// 20.11.6.6 time_point comparisons
template <class Clock, class Duration1, class Duration2>
   bool operator==(const time_point<Clock, Duration1>& lhs,
```

§ 20.11.2 579

```
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);
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);
// 20.11.6.7, time_point_cast
template <class ToDuration, class Clock, class Duration>
  time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);
// 20.11.7, clocks
class system_clock;
class steady_clock;
class high_resolution_clock;
} // namespace chrono
} // namespace std
```

#### 20.11.3 Clock requirements

[time.clock.req]

- A clock is a bundle consisting of a duration, a time\_point, and a function now() to get the current time\_point. The origin of the clock's time\_point is referred to as the clock's *epoch*. A clock shall meet the requirements in Table 59.
- In Table 59 C1 and C2 denote clock types. t1 and t2 are values returned by C1::now() where the call returning t1 happens before (1.10) the call returning t2 and both of these calls occur before C1::time\_point::max(). [Note: this means C1 did not wrap around between t1 and t2. —end note]

Expression	Return type	Operational semantics	
C1::rep	An arithmetic type or a class	The representation type of	
	emulating an arithmetic type	C1::duration.	
C1::period	a specialization of ratio	The tick period of the clock in	
		seconds.	
C1::duration	chrono::duration <c1::rep,< td=""><td>The duration type of the</td></c1::rep,<>	The duration type of the	
	C1::period>	clock.	
C1::time_point	chrono::time_point <c1> or</c1>	The time_point type of the	
	chrono::time_point <c2,< td=""><td>clock. <math>C1</math> and <math>C2</math> shall refer to</td></c2,<>	clock. $C1$ and $C2$ shall refer to	
	C1::duration>	the same epoch.	

Table 59 — Clock requirements

§ 20.11.3 580

Expression	Return type	Operational semantics	
C1::is_steady	const bool	true if t1 <= t2 is always	
		true and the time between	
		clock ticks is constant,	
		otherwise false.	
C1::now()	C1::time_point	Returns a time_point object	
		representing the current point	
		in time	

Table 59 — Clock requirements (continued)

- <sup>3</sup> [Note: The relative difference in durations between those reported by a given clock and the SI definition is a measure of the quality of implementation. end note]
- 4 A type TC meets the TrivialClock requirements if:
  - TC satisfies the Clock requirements (20.11.3),
  - the types TC::rep, TC::duration, and TC::time\_point satisfy the requirements of EqualityComparable (Table 17), LessThanComparable (Table 18), DefaultConstructible (Table 19), CopyConstructible (Table 21), CopyAssignable (Table 23), Destructible (Table 24), and the requirements of numeric types (26.2). [Note: this means, in particular, that operations on these types will not throw exceptions. end note]
  - Ivalues of the types TC::rep, TC::duration, and TC::time\_point are swappable (17.6.3.2),
  - the function TC::now() does not throw exceptions, and
  - the type TC::time\_point::clock meets the TrivialClock requirements, recursively.

#### 20.11.4 Time-related traits

[time.traits]

### 20.11.4.1 treat\_as\_floating\_point

[time.traits.is\_fp]

```
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 implicit conversions are allowed among durations. Otherwise, the implicit convertibility depends on the tick periods of the durations. [Note: The intention of this trait is to indicate whether a given class behaves like a floating-point type, and thus allows division of one value by another with acceptable loss of precision. If treat\_as\_floating\_point<Rep>::value is false, Rep will be treated as if it behaved like an integral type for the purpose of these conversions. — end note]

### 20.11.4.2 duration\_values

 $[time.traits.duration\_values]$ 

```
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 durations representation (Rep). This is done because the representation might be a class type with behavior which requires some other implementation to return these special values. In that case, the author of that class type should specialize duration\_values to return the indicated values.

§ 20.11.4.2 581

```
static constexpr Rep zero();
2
        Returns: Rep(0). [Note: Rep(0) is specified instead of Rep() because Rep() may have some other
        meaning, such as an uninitialized value. — end note
3
        Remark: The value returned shall be the additive identity.
   static constexpr Rep min();
4
        Returns: numeric_limits<Rep>::lowest().
5
        Remark: The value returned shall compare less than or equal to zero().
   static constexpr Rep max();
6
        Returns: numeric_limits<Rep>::max().
        Remark: The value returned shall compare greater than zero().
   20.11.4.3 Specializations of common_type
                                                                           [time.traits.specializations]
   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 com-
   mon 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.
   20.11.5
             Class template duration
                                                                                      [time.duration]
  A duration type measures time between two points in time (time_points). A duration has a representation
   which holds a count of ticks and a tick period. The tick period is the amount of time which occurs from one
   tick to the next, in units of seconds. It is expressed as a rational constant using the template ratio.
     template <class Rep, class Period = ratio<1>>
     class duration {
```

§ 20.11.5

public:

private:

public:

typedef Rep

typedef Period period;

rep rep\_; // exposition only

rep;

// 20.11.5.1, construct/copy/destroy:
constexpr duration() = default;

```
template <class Rep2>
        constexpr explicit duration(const Rep2& r);
    template <class Rep2, class Period2>
       constexpr duration(const duration<Rep2, Period2>& d);
    ~duration() = default;
    duration(const duration&) = default;
    duration& operator=(const duration&) = default;
    // 20.11.5.2, observer:
    constexpr rep count() const;
    // 20.11.5.3, arithmetic:
    constexpr duration operator+() const;
    constexpr duration operator-() const;
    duration& operator++();
    duration operator++(int);
    duration& operator--();
    duration operator -- (int);
    duration& operator+=(const duration& d);
    duration& operator = (const duration& d);
    duration& operator*=(const rep& rhs);
    duration& operator/=(const rep& rhs);
    duration& operator%=(const rep& rhs);
    duration& operator%=(const duration& rhs);
    // 20.11.5.4, 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
                                        // holds a count of milliseconds using a long long
  duration<long long, milli> d1;
  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]
20.11.5.1 duration constructors
                                                                                  [time.duration.cons]
template <class Rep2>
  constexpr explicit duration(const Rep2& r);
§ 20.11.5.1
                                                                                                     583
```

2

3

4

5

6

1

```
Remarks: This constructor shall not participate in overload resolution unless Rep2 is implicitly con-
        vertible to rep and
          — treat_as_floating_point<rep>::value is true or
          — treat_as_floating_point<Rep2>::value is false.
        [Example:
          duration<int, milli> d(3);
                                               // OK
          duration<int, milli> d(3.5);
                                              // error
         — end example]
2
        Effects: Constructs an object of type duration.
3
        Postcondition: count() == static_cast<rep>(r).
   template <class Rep2, class Period2>
     constexpr duration(const duration<Rep2, Period2>& d);
        Remarks: This constructor shall not participate in overload resolution unless treat_as_floating_-
        point<rep>::value is true or both ratio_divide<Period2, period>::den is 1 and treat_as_-
        floating_point<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 ] [Example:
          duration<int, milli> ms(3);
          duration<int, micro> us = ms;
                                               // OK
          duration<int, milli> ms2 = us;
                                               // error
         — end example]
5
        Effects: Constructs an object of type duration, constructing rep_ from
        duration_cast<duration>(d).count().
   20.11.5.2 duration observer
                                                                             [time.duration.observer]
   constexpr rep count() const;
1
        Returns: rep_.
   20.11.5.3 duration arithmetic
                                                                           [time.duration.arithmetic]
   constexpr duration operator+() const;
        Returns: *this.
   constexpr duration operator-() const;
2
        Returns: duration(-rep_);.
   duration& operator++();
3
        Effects: ++rep .
4
        Returns: *this.
```

§ 20.11.5.3 584

```
duration operator++(int);
         Returns: duration(rep_++);.
    duration& operator--();
6
         Effects: --rep_.
7
         Returns: *this.
   duration operator--(int);
8
         Returns: duration(rep_--);.
    duration& operator+=(const duration& d);
9
         Effects: rep_ += d.count().
10
         Returns: *this.
   duration& operator-=(const duration& d);
11
         Effects: rep_ -= d.count().
12
         Returns: * this.
   duration& operator*=(const rep& rhs);
13
         Effects: rep_ *= rhs.
         Returns: *this.
    duration& operator/=(const rep& rhs);
15
         Effects: rep_ /= rhs.
16
         Returns: *this.
   duration& operator%=(const rep& rhs);
17
         Effects: rep_ %= rhs.
18
         Returns: *this.
   duration& operator%=(const duration& rhs);
19
         Effects: rep_ %= rhs.count().
20
         Returns: *this.
```

§ 20.11.5.3 585

```
20.11.5.4 duration special values
                                                                                [time.duration.special]
   static constexpr duration zero();
1
        Returns: duration(duration_values<rep>::zero()).
   static constexpr duration min();
        Returns: duration(duration_values<rep>::min()).
   static constexpr duration max();
3
        Returns: duration(duration_values<rep>::max()).
   20.11.5.5 duration non-member arithmetic
                                                                          [time.duration.nonmember]
<sup>1</sup> In the function descriptions that follow, CD represents the return type of the function. CR(A,B) represents
   common_type<A, B>::type.
   template <class Rep1, class Period1, class Rep2, class Period2>
     constexpr typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2> >::type
     operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
        Returns: CD(CD(lhs).count() + CD(rhs).count()).
   template <class Rep1, class Period1, class Rep2, class Period2>
     constexpr typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2> >::type
     operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
        Returns: CD(CD(lhs).count() - CD(rhs).count()).
   template <class Rep1, class Period, class Rep2>
     constexpr duration<typename common_type<Rep1, Rep2>::type, Period>
     operator*(const duration<Rep1, Period>& d, const Rep2& s);
4
        Remarks: This operator shall not participate in overload resolution unless Rep2 is implicitly convertible
        to CR(Rep1, Rep2).
5
        Returns: CD(CD(d).count() * s).
   template <class Rep1, class Rep2, class Period>
     constexpr duration<typename common_type<Rep1, Rep2>::type, Period>
     operator*(const Rep1& s, const duration<Rep2, Period>& d);
6
        Remarks: This operator shall not participate in overload resolution unless Rep1 is implicitly convertible
        to CR(Rep1, Rep2).
        Returns: d * s.
   template <class Rep1, class Period, class Rep2>
     constexpr duration<typename common_type<Rep1, Rep2>::type, Period>
     operator/(const duration<Rep1, Period>& d, const Rep2& s);
```

§ 20.11.5.5

```
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.
9
         Returns: CD(CD(d).count() / s).
   template <class Rep1, class Period1, class Rep2, class Period2>
      constexpr typename common_type<Rep1, Rep2>::type
      operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
10
         Returns: CD(lhs).count() / CD(rhs).count().
   template <class Rep1, class Period, class Rep2>
      constexpr duration<typename common_type<Rep1, Rep2>::type, Period>
      operator%(const duration<Rep1, Period>& d, const Rep2& s);
11
         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.
12
         Returns: CD(CD(d).count() \% s).
   template <class Rep1, class Period1, class Rep2, class Period2>
      constexpr typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
      operator%(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
13
         Returns: CD(CD(lhs).count() \% CD(rhs).count()).
   20.11.5.6 duration comparisons
                                                                           [time.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>
      constexpr bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
2
         Returns: CT(lhs).count() == CT(rhs).count().
   template <class Rep1, class Period1, class Rep2, class Period2>
      constexpr bool operator!=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
         Returns: !(lhs == rhs).
   template <class Rep1, class Period1, class Rep2, class Period2>
      constexpr bool operator<(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
         Returns: CT(lhs).count() < CT(rhs).count().
   template <class Rep1, class Period1, class Rep2, class Period2>
      constexpr bool operator<=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
         Returns: !(rhs < lhs).
   template <class Rep1, class Period1, class Rep2, class Period2>
     constexpr bool operator>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
   § 20.11.5.6
                                                                                                      587
```

```
6
        Returns: rhs < lhs.
   template <class Rep1, class Period1, class Rep2, class Period2>
     constexpr bool operator>=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
        Returns: !(lhs < rhs).
   20.11.5.7 duration_cast
                                                                                   [time.duration.cast]
   template <class ToDuration, class Rep, class Period>
     constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
1
        Remarks: This function shall not participate in overload resolution unless ToDuration is an instanti-
        ation of duration.
2
        Returns: Let CF be ratio_divide<Period, typename ToDuration::period>, and CR be common_-
        type< typename ToDuration::rep, Rep, intmax_t>::type.
         - If CF::num == 1 and CF::den == 1, returns
               ToDuration(static_cast<typename ToDuration::rep>(d.count()))
          — otherwise, if CF::num != 1 and CF::den == 1, returns
               ToDuration(static_cast<typename ToDuration::rep>(
                 static_cast<CR>(d.count()) * static_cast<CR>(CF::num)))
          — otherwise, if CF::num == 1 and CF::den != 1, returns
               ToDuration(static_cast<typename ToDuration::rep>(
                 static_cast<CR>(d.count()) / static_cast<CR>(CF::den)))
          — otherwise, returns
               ToDuration(static_cast<typename ToDuration::rep>(
                 static_cast<CR>(d.count()) * static_cast<CR>(CF::num) / static_cast<CR>(CF::den)))
        Notes: This function does not use any implicit conversions; all conversions are done with static_cast.
        It avoids multiplications and divisions when it is known at compile time that one or more arguments
        is 1. Intermediate computations are carried out in the widest representation and only converted to the
        destination representation at the final step.
                                                                                          [time.point]
            Class template time_point
     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
```

§ 20.11.6 588

public:

// 20.11.6.1, 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);
       // 20.11.6.2, observer:
       duration time_since_epoch() const;
       // 20.11.6.3, arithmetic:
       time_point& operator+=(const duration& d);
       time_point& operator-=(const duration& d);
       // 20.11.6.4, special values:
       static constexpr time_point min();
       static constexpr time_point max();
     };
<sup>1</sup> Clock shall meet the Clock requirements (20.11.7).
<sup>2</sup> If Duration is not an instance of duration, the program is ill-formed.
                                                                                         [time.point.cons]
   20.11.6.1 time_point constructors
   time_point();
        Effects: Constructs an object of type time_point, initializing d_ with duration::zero(). Such a
1
        time_point object represents the epoch.
   time_point(const duration& d);
2
        Effects: Constructs an object of type time_point, initializing d_ 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.
         Effects: Constructs an object of type time_point, initializing d_ with t.time_since_epoch().
   20.11.6.2 time_point observer
                                                                                    [time.point.observer]
   duration time_since_epoch() const;
        Returns: d_.
                                                                                  [time.point.arithmetic]
   20.11.6.3 time_point arithmetic
   time_point& operator+=(const duration& d);
1
        Effects: d_ += d.
2
        Returns: *this.
   time_point& operator-=(const duration& d);
3
        Effects: d_- = d.
4
        Returns: *this.
   § 20.11.6.3
                                                                                                        589
```

```
20.11.6.4 time_point special values
                                                                               [time.point.special]
static constexpr time_point min();
     Returns: time_point(duration::min()).
static constexpr time_point max();
     Returns: time_point(duration::max()).
20.11.6.5 time_point non-member arithmetic
                                                                         [time.point.nonmember]
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: 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().
20.11.6.6 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().</pre>
```

§ 20.11.6.6 590

```
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).
                                                                                       [time.point.cast]
   20.11.6.7 time_point_cast
   template <class ToDuration, class Clock, class Duration>
     time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);
1
        Remarks: This function shall not participate in overload resolution unless ToDuration is an instanti-
        ation of duration.
        Returns: time_point<Clock, ToDuration>(duration_cast<ToDuration>(t.time_since_epoch())).
   20.11.7 Clocks
                                                                                           [time.clock]
<sup>1</sup> The types defined in this subclause shall satisfy the TrivialClock requirements (20.11.3).
   20.11.7.1 Class system_clock
                                                                                    [time.clock.system]
Objects of class system_clock represent wall clock time from the system-wide realtime clock.
     class system_clock {
     public:
       typedef see below
                                                    rep;
       {\tt typedef\ ratio} {\it <} unspecified\ ,\ unspecified\ >
                                                     period;
       typedef chrono::duration<rep, period>
                                                    duration;
       typedef chrono::time_point<system_clock>
                                                    time_point;
       static const bool is_steady =
                                                 unspecified;
       static time_point now() noexcept;
       // Map to C API
       static time_t
                          to_time_t (const time_point& t) noexcept;
       static time_point from_time_t(time_t t) noexcept;
     };
   typedef unspecified system_clock::rep;
        Requires: system_clock::duration::min() < system_clock::duration::zero() shall be true.
        [Note: This implies that rep is a signed type. -end note]
   static time_t to_time_t(const time_point& t) noexcept;
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.
```

§ 20.11.7.1 591

```
static time_point from_time_t(time_t t) noexcept;
```

Returns: A time\_point object that represents the same point in time as t when both values are restricted to the coarser of the precisions of time\_t and time\_point. It is implementation defined whether values are rounded or truncated to the required precision.

#### 20.11.7.2 Class steady\_clock

[time.clock.steady]

Objects of class steady\_clock represent clocks for which values of time\_point never decrease as physical time advances and for which values of time\_point advance at a steady rate relative to real time. That is, the clock may not be adjusted.

#### 20.11.7.3 Class high\_resolution\_clock

[time.clock.hires]

Objects of class high\_resolution\_clock represent clocks with the shortest tick period. high\_resolution\_-clock may be a synonym for system\_clock or steady\_clock.

### 20.11.8 Date and time functions

[date.time]

Table 60 describes the header <ctime>.

Table 60 — Header <ctime> synopsis

Type	$\mathbf{Name}(\mathbf{s})$			
Macros:	NULL	CLOCKS_PER_SEC		
Types:	size_t	clock_t	time_t	
Struct:	tm			
Functions:				
asctime	clock	difftime	localtime	strftime
ctime	gmtime	mktime	time	

<sup>2</sup> The contents are the same as the Standard C library header <time.h>.<sup>231</sup> The functions asctime, ctime, gmtime, and localtime are not required to avoid data races (17.6.5.9).

SEE ALSO: ISO C Clause 7.12, Amendment 1 Clause 4.6.4.

§ 20.11.8 592

<sup>231)</sup> 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 O.

### 20.12 Class template scoped\_allocator\_adaptor

[allocator.adaptor]

### 20.12.1 Header <scoped\_allocator> synopsis

[allocator.adaptor.syn]

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]

```
namespace std {
  template <class OuterAlloc, class... InnerAllocs>
    class scoped_allocator_adaptor : public OuterAlloc {
    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;
     };
```

§ 20.12.1 593

```
scoped_allocator_adaptor();
  template <class OuterA2>
    scoped_allocator_adaptor(OuterA2&& outerAlloc,
                             const InnerAllocs&... innerAllocs) noexcept;
  scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;
  scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;
  template <class OuterA2>
    scoped_allocator_adaptor(
      const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other) noexcept;
  template <class OuterA2>
    scoped_allocator_adaptor(
      scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;
  ~scoped_allocator_adaptor();
  inner_allocator_type& inner_allocator() noexcept;
  const inner_allocator_type& inner_allocator() const noexcept;
  outer_allocator_type& outer_allocator() noexcept;
  const outer_allocator_type& outer_allocator() const noexcept;
 pointer allocate(size_type n);
 pointer allocate(size_type n, const_void_pointer hint);
  void deallocate(pointer p, size_type n);
  size_type max_size() const;
  template <class T, class... Args>
    void construct(T* p, Args&& args);
  template <class T1, class T2, class... Args1, class... Args2>
    void construct(pair<T1, T2>* p, piecewise_construct_t,
                   tuple<Args1...> x, tuple<Args2...> y);
  template <class T1, class T2>
    void construct(pair<T1, T2>* p);
  template <class T1, class T2, class U, class V>
    void construct(pair<T1, T2>* p, U&& x, V&& y);
  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 < Outer A1, Inner Allocs... > & a,
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
```

§ 20.12.1 594

}

[allocator.adaptor.types]

595

Scoped allocator adaptor member types

§ 20.12.3

```
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;
        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;
3
        Type: true_type if allocator_traits<A>::propagate_on_container_move_assignment::value is
        true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false type.
   typedef see below propagate_on_container_swap;
        Type: true_type if allocator_traits<A>::propagate_on_container_swap::value is true for any
4
        A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.
   20.12.3
                                                                           [allocator.adaptor.cnstr]
             Scoped allocator adaptor constructors
   scoped_allocator_adaptor();
1
        Effects: value-initializes the OuterAlloc base class and the inner allocator object.
   template <class OuterA2>
     scoped_allocator_adaptor(OuterA2&& outerAlloc,
                              const InnerAllocs&... innerAllocs) noexcept;
2
        Requires: OuterAlloc shall be constructible from OuterA2.
3
        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 corre-
        sponding allocator from the argument list).
   scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;
4
        Effects: initializes each allocator within the adaptor with the corresponding allocator from other.
   scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;
5
        Effects: move constructs each allocator within the adaptor with the corresponding allocator from
        other.
   template <class OuterA2>
     scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2,
                                                              InnerAllocs...>& other) noexcept;
```

```
Requires: OuterAlloc shall be constructible from OuterA2.
7
        Effects: initializes each allocator within the adaptor with the corresponding allocator from other.
  template <class OuterA2>
     scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2,</pre>
                                                      InnerAllocs...>&& other) noexcept;
8
        Requires: OuterAlloc shall be constructible from OuterA2.
9
        Effects: initializes each allocator within the adaptor with the corresponding allocator rvalue from
        other.
  20.12.4
             Scoped allocator adaptor members
                                                                     [allocator.adaptor.members]
  In the construct member functions, OUTERMOST(x) is x if x does not have an outer_allocator() mem-
  ber 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() noexcept;
  const inner_allocator_type& inner_allocator() const noexcept;
        Returns: *this if sizeof...(InnerAllocs) is zero; otherwise, inner.
  outer_allocator_type& outer_allocator() noexcept;
        Returns: static cast<OuterAlloc&>(*this).
  const outer_allocator_type& outer_allocator() const noexcept;
        Returns: static_cast<const OuterAlloc&>(*this).
  pointer allocate(size_type n);
5
        Returns: allocator_traits<OuterAlloc>::allocate(outer_allocator(), n).
  pointer allocate(size_type n, const_void_pointer hint);
6
        Returns: allocator_traits<OuterAlloc>::allocate(outer_allocator(), n, hint).
  void deallocate(pointer p, size_type n) noexcept;
7
        Effects: allocator_traits<OuterAlloc>::deallocate(outer_allocator(), p, n);
  size_type max_size() const;
        Returns: allocator_traits<OuterAlloc>::max_size(outer_allocator()).
```

596

§ 20.12.4

```
template <class T, class... Args>
void construct(T* p, Args&&... args);
```

9 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 OUT-ERMOST\_ALLOC\_TRAITS(\*this)::construct(OUTERMOST(\*this), p, allocator\_arg, inner\_allocator(), std::forward<Args>(args)...).
- Otherwise, if uses\_allocator<T, inner\_allocator\_type>::value is true and is\_construct-ible<T, Args..., inner\_allocator\_type>::value is true, calls OUTERMOST\_ALLOC\_-TRAITS(\*this):: construct(OUTERMOST(\*this), p, std::forward<Args>(args)..., inner\_allocator()).
- 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]

- 10 Requires: all of the types in Args1 and Args2 shall be CopyConstructible (Table 21).
- 11 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\_construct-ible<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\_construct-ible<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\_construct-ible<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\_construct-ible<T2, Args2..., inner\_allocator\_type>::value is true, then yprime is tuple\_cat(y, tuple<inner\_allocator\_type&>(inner\_allocator\_type())).

§ 20.12.4 597

```
— 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);
12
         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);
13
         Effects: equivalent to this->construct(p, piecewise_construct, forward_as_tuple(std::for-
        ward<U>(x)), forward_as_tuple(std::forward<V>(y))).
   template <class T1, class T2, class U, class V>
     void construct(pair<T1, T2>* p, const pair<U, V>& x);
14
         Effects: equivalent to this->construct(p, piecewise_construct, forward_as_tuple(x.first),
        forward_as_tuple(x.second)).
   template <class T1, class T2, class U, class V>
     void construct(pair<T1, T2>* p, pair<U, V>&& x);
         Effects: equivalent to this->construct(p, piecewise_construct, forward_as_tuple(std::for-
        ward<U>(x.first)), forward_as_tuple(std::forward<V>(x.second))).
   template <class T>
     void destroy(T* p);
16
         Effects: calls OUTERMOST_ALLOC_TRAITS(*this)::destroy(OUTERMOST(*this), p).
   scoped_allocator_adaptor select_on_container_copy_construction() const;
17
         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.
                                                                       [scoped.adaptor.operators]
   20.12.5
             Scoped allocator operators
   template <class OuterA1, class OuterA2, class... InnerAllocs>
     bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                     const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
         Returns: a.outer_allocator() == b.outer_allocator() if sizeof...(InnerAllocs) is zero; oth-
        erwise, a.outer_allocator() == b.outer_allocator() && a.inner_allocator() == b.inner_-
        allocator().
   template <class OuterA1, class OuterA2, class... InnerAllocs>
     bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                     const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
         Returns: !(a == b).
   § 20.12.5
                                                                                                  598
```

```
20.13 Class type_index
                                                                                        [type.index]
                                                                             [type.index.synopsis]
20.13.1
          Header <typeindex> synopsis
  namespace std {
    class type_index;
    template <class T> struct hash;
    template<> struct hash<type_index>;
20.13.2
                                                                            [type.index.overview]
          type_index overview
  namespace std {
    class type_index {
    public:
      type_index(const type_info& rhs) noexcept;
      bool operator==(const type_index& rhs) const noexcept;
      bool operator!=(const type_index& rhs) const noexcept;
      bool operator< (const type_index& rhs) const noexcept;</pre>
      bool operator<= (const type_index& rhs) const noexcept;</pre>
      bool operator> (const type_index& rhs) const noexcept;
      bool operator>= (const type_index& rhs) const noexcept;
      size_t hash_code() const;
      const char* name() const;
    private:
                                  // exposition only
      const type_info* target;
      // Note that the use of a pointer here, rather than a reference,
      // means that the default copy/move constructor and assignment
      // operators will be provided and work as expected.
    };
The class type_index provides a simple wrapper for type_info which can be used as an index type in
associative containers (23.4) and in unordered associative containers (23.5).
20.13.3 type_index members
                                                                            [type.index.members]
type_index(const type_info& rhs) noexcept;
     Effects: constructs a type_index object, the equivalent of target = &rhs.
bool operator==(const type_index& rhs) const noexcept;
      Returns: *target == *rhs.target
bool operator!=(const type_index& rhs) const noexcept;
     Returns: *target != *rhs.target
bool operator<(const type_index& rhs) const noexcept;</pre>
     Returns: target->before(*rhs.target)
bool operator<=(const type_index& rhs) const noexcept;</pre>
     Returns: !rhs.target->before(*target)
§ 20.13.3
                                                                                                  599
```

2

3

4

```
bool operator>(const type_index& rhs) const noexcept;
        Returns: rhs.target->before(*target)
6
  bool operator>=(const type_index& rhs) const noexcept;
        Returns: !target->before(*rhs.target)
  size_t hash_code() const;
        Returns: target->hash_code()
  const char* name() const;
9
        Returns: target->name()
  20.13.4 Hash support
                                                                               [type.index.hash]
```

template <> struct hash<type\_index>;

Requires: the template specialization shall meet the requirements of class template hash (20.8.12). For an object index of type type\_index, hash<type\_index>()(index) shall evaluate to the same result as index.hash\_code().

§ 20.13.4 600

## 21 Strings library

# [strings]

21.1 General [strings.general]

<sup>1</sup> 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*.

<sup>2</sup> The following subclauses describe a character traits class, a string class, and null-terminated sequence utilities, as summarized in Table 61.

	Subclause	Header(s)
21.2	Character traits	<string></string>
21.3	String classes	<string></string>
		<cctype></cctype>
		<cwctype></cwctype>
21.7	Null-terminated sequence utilities	<cstring></cstring>
		<cwchar></cwchar>
		<cstdlib></cstdlib>
		<cuchar></cuchar>

Table 61 — Strings library summary

## 21.2 Character traits

[char.traits]

- This subclause defines requirements on classes representing *character traits*, and defines a class template char\_traits<charT>, along with four specializations, char\_traits<char>, char\_traits<char16\_t>, char\_traits<char32\_t>, and char\_traits<wchar\_t>, that satisfy those requirements.
- <sup>2</sup> 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.
- 4 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

[char.traits.require]

In Table 62, X denotes a Traits class defining types and functions for the character container type CharT; c and d denote values of type CharT; p and q denote values of type const CharT\*; s denotes a value of type CharT\*; n, i and j denote values of type std::size\_t; e and f denote values of type X::int\_type; pos denotes a value of type X::pos\_type; state denotes a value of type X::state\_type; and r denotes an lvalue of type CharT. Operations on Traits shall not throw exceptions.

§ 21.2.1 601

 $\odot$  ISO/IEC N3337

Table 62 — Character traits requirements

Expression	Return type	$\begin{array}{c} {\bf Assertion/note} \\ {\bf pre-/post-condition} \end{array}$	Complexity
X::char_type	charT	(described in 21.2.2)	compile-time
X::int_type		(described in 21.2.2)	compile-time
X::off_type		(described in 21.2.2)	compile-time
X::pos_type		(described in 21.2.2)	compile-time
X::state_type		(described in 21.2.2)	compile-time
X::eq(c,d)	bool	yields: whether c is to be	constant
• .		treated as equal to d.	
X::lt(c,d)	bool	yields: whether <b>c</b> is to be	constant
		treated as less than d.	
X::compare(p,q,n)	int	yields: 0 if for each i in [0,n),	linear
		X::eq(p[i],q[i]) is true; else,	
		a negative value if, for some <b>j</b>	
		in [0,n), X::lt(p[j],q[j]) is	
		true and for each i in [0,j)	
		X::eq(p[i],q[i]) is true; else	
		a positive value.	
X::length(p)	std::size_t	yields: the smallest i such that	linear
		X::eq(p[i],charT()) is true.	
<pre>X::find(p,n,c)</pre>	<pre>const X::char_type*</pre>	yields: the smallest q in	linear
		[p,p+n) such that	
		X::eq(*q,c) is true, zero	
		otherwise.	
X::move(s,p,n)	<pre>X::char_type*</pre>	for each i in [0,n), performs	linear
		X::assign(s[i],p[i]).	
		Copies correctly even where the	
		ranges [p,p+n) and [s,s+n)	
X::copy(s,p,n)	X::char_type*	overlap. yields: s. pre: p not in [s,s+n). yields:	linear
xcopy(s,p,n)	xchar_cype*	s. for each i in [0,n), performs	imeai
		X::assign(s[i],p[i]).	
X::assign(r,d)	(not used)	assigns r=d.	constant
X::assign(s,n,c)	X::char_type*	for each i in [0,n), performs	linear
	III. OHUI _ Jypo .	X::assign(s[i],c). yields: s.	1111001
X::not_eof(e)	int_type	yields: e if	constant
	JF-	<pre>X::eq_int_type(e,X::eof())</pre>	V
		is false, otherwise a value f	
		such that	
		<pre>X::eq_int_type(f,X::eof())</pre>	
		is false.	
X::to_char_type(e)	X::char_type	yields: if for some c,	constant
		X::eq_int_type(e,X::to	
		<pre>int_type(c)) is true, c; else</pre>	
		some unspecified value.	

§ 21.2.1 602

Expression Return type Assertion/note pre-/post-condition		Complexity	
X::to_int_type(c)	X::int_type	yields: some value e, constrained by the definitions of to_char_type and eq_int_type.	constant
X::eq_int_type(e,f)	bool	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.	constant
X::eof()	X::int_type	<pre>yields: a value e such that X::eq_int_type(e,X::to int_type(c)) is false for all values c.</pre>	constant

Table 62 — Character traits requirements (continued)

<sup>2</sup> The struct template

1

template<class charT> struct char\_traits;

shall be provided in the header <string> as a basis for explicit specializations.

## 21.2.2 traits typedefs

[char.traits.typedefs]

typedef CHAR\_T char\_type;

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;
```

Requires: For a certain character container type char\_type, a related container type INT\_T shall be a type or class which can represent all of the valid characters converted from the corresponding char\_type values, as well as an end-of-file value, eof(). The type int\_type represents a character container type which can hold end-of-file to be used as a return type of the iostream class member functions. 232

```
typedef implementation-defined off_type;
typedef implementation-defined pos_type;
```

3 Requires: Requirements for off\_type and pos\_type are described in 27.2.2 and 27.3.

typedef STATE\_T state\_type;

Requires: state\_type shall meet the requirements of CopyAssignable (Table 23), CopyConstructible (Table 21), and DefaultConstructible (Table 19) types.

§ 21.2.2 603

<sup>232)</sup> If eof() can be held in char\_type then some iostreams operations may give surprising results.

```
21.2.3 char_traits specializations
```

[char.traits.specializations]

```
namespace std {
  template<> struct char_traits<char>;
  template<> struct char_traits<char16_t>;
  template<> struct char_traits<char32_t>;
  template<> struct char_traits<wchar_t>;
}
```

- 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>.
- <sup>2</sup> The requirements for the members of these specializations are given in Clause 21.2.1.

#### 21.2.3.1 struct char\_traits<char>

[char.traits.specializations.char]

```
namespace std {
  template<> struct char_traits<char> {
    typedef char
                        char_type;
    typedef int
                        int_type;
    typedef streamoff off_type;
    typedef streampos
                        pos_type;
    typedef mbstate_t
                        state_type;
    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;
    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);
    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
 };
}
```

- The defined types for int\_type, pos\_type, off\_type, and state\_type shall be int, streampos, streamoff, and mbstate\_t respectively.
- <sup>2</sup> The type streampos shall be an implementation-defined type that satisfies the requirements for pos\_type in 27.2.2 and 27.3.
- <sup>3</sup> The type streamoff shall be an implementation-defined type that satisfies the requirements for off\_type in 27.2.2 and 27.3.
- <sup>4</sup> 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.
- 6 The member eof() shall return EOF.

```
21.2.3.2 struct char traits<char16 t>
```

[char.traits.specializations.char16 t]

§ 21.2.3.2

```
namespace std {
  template<> struct char_traits<char16_t> {
    typedef char16_t
                     char_type;
    typedef uint_least16_t int_type;
    typedef streamoff
                         off_type;
    typedef u16streampos
                           pos_type;
    typedef mbstate_t
                           state_type;
   static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;
    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
   static const char_type* find(const char_type* s, size_t n,
                                const char_type& a);
   static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);
   static constexpr int_type not_eof(int_type c) noexcept;
   static constexpr char_type to_char_type(int_type c) noexcept;
   static constexpr int_type to_int_type(char_type c) noexcept;
   static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
   static constexpr int_type eof() noexcept;
}
```

- <sup>1</sup> The type u16streampos shall be an implementation-defined type that satisfies the requirements for pos\_type in 27.2.2 and 27.3.
- <sup>2</sup> The two-argument members assign, eq, and lt shall be defined identically to the built-in operators =, ==, and < respectively.
- <sup>3</sup> 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> [char.traits.specializations.char32_t]
```

```
namespace std {
  template<> struct char_traits<char32_t> {
    typedef char32_t
                           char_type;
    typedef uint_least32_t int_type;
    typedef streamoff
                           off_type;
    typedef u32streampos
                           pos_type;
    typedef mbstate_t
                            state_type;
    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;
    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);
```

§ 21.2.3.3

```
static constexpr int_type not_eof(int_type c) noexcept;
static constexpr char_type to_char_type(int_type c) noexcept;
static constexpr int_type to_int_type(char_type c) noexcept;
static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
static constexpr int_type eof() noexcept;
};
}
```

- <sup>1</sup> The type u32streampos shall be an implementation-defined type that satisfies the requirements for pos\_type in 27.2.2 and 27.3.
- <sup>2</sup> The two-argument members assign, eq, and lt shall be defined identically to the built-in operators =, ==, and < respectively.
- <sup>3</sup> The member eof() shall return an implementation-defined constant that cannot appear as a Unicode code point.

#### 21.2.3.4 struct char traits<wchar t>

[char.traits.specializations.wchar.t]

```
namespace std {
  template<> struct char_traits<wchar_t> {
                        char_type;
    typedef wchar_t
    typedef wint_t
                         int_type;
    typedef streamoff
                        off_type;
    typedef wstreampos
                         pos_type;
    typedef mbstate_t
                         state_type;
    static void assign(char_type& c1, const char_type& c2) noexcept;
    static constexpr bool eq(char_type c1, char_type c2) noexcept;
    static constexpr bool lt(char_type c1, char_type c2) noexcept;
    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);
    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
  };
}
```

- The defined types for int\_type, pos\_type, and state\_type shall be wint\_t, wstreampos, and mbstate\_t respectively.
- <sup>2</sup> The type wstreampos shall be an implementation-defined type that satisfies the requirements for pos\_type in 27.2.2 and 27.3.
- <sup>3</sup> 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.
- <sup>4</sup> The two-argument members assign, eq, and lt shall be defined identically to the built-in operators =, ==, and < respectively.

§ 21.2.3.4 606

<sup>5</sup> The member eof() shall return WEOF.

## 21.3 String classes

[string.classes]

The header <string> defines the basic\_string class template for manipulating varying-length sequences of char-like objects and four typedefs, string, u16string, u32string, and wstring, that name the specializations basic\_string<char>, basic\_string<char16\_t>, basic\_string<char32\_t>, and basic\_string< wchar\_t>, respectively.

```
Header <string> synopsis
```

```
#include <initializer_list>
namespace std {
 // 21.2, character traits:
 template<class charT> struct char_traits;
 template <> struct char_traits<char>;
 template <> struct char_traits<char16_t>;
 template <> struct char_traits<char32_t>;
 template <> struct char_traits<wchar_t>;
 // 21.4, basic string:
 template<class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
      class basic_string;
 template < class charT, class traits, class Allocator >
   basic_string<charT,traits,Allocator>
      operator+(const basic_string<charT,traits,Allocator>& lhs,
                const basic_string<charT, traits, Allocator>& rhs);
 template < class charT, class traits, class Allocator>
   basic_string<charT,traits,Allocator>
      operator+(basic_string<charT,traits,Allocator>&& lhs,
                const basic_string<charT,traits,Allocator>& rhs);
 template < class charT, class traits, class Allocator >
   basic_string<charT,traits,Allocator>
      operator+(const basic_string<charT,traits,Allocator>& lhs,
                basic_string<charT,traits,Allocator>&& rhs);
 template < class charT, class traits, class Allocator >
    basic_string<charT,traits,Allocator>
      operator+(basic_string<charT,traits,Allocator>&& lhs,
                basic_string<charT,traits,Allocator>&& rhs);
 template < class charT, class traits, class Allocator >
   basic_string<charT,traits,Allocator>
      operator+(const charT* lhs,
                const basic_string<charT,traits,Allocator>& rhs);
 template<class charT, class traits, class Allocator>
   basic_string<charT,traits,Allocator>
      operator+(const charT* lhs,
                basic_string<charT,traits,Allocator>&& rhs);
 template < class charT, class traits, class Allocator >
    basic_string<charT,traits,Allocator>
      operator+(charT lhs, const basic_string<charT,traits,Allocator>& rhs);
 template < class charT, class traits, class Allocator >
   basic_string<charT,traits,Allocator>
      operator+(charT lhs, basic_string<charT,traits,Allocator>&& rhs);
 template < class charT, class traits, class Allocator >
```

```
basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              const charT* rhs);
template < class charT, class traits, class Allocator >
  basic_string<charT,traits,Allocator>
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              const charT* rhs);
template < class charT, class traits, class Allocator >
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs, charT rhs);
template < class charT, class traits, class Allocator >
  basic_string<charT,traits,Allocator>
    operator+(basic_string<charT,traits,Allocator>&& lhs, charT rhs);
template < class charT, class traits, class Allocator >
  bool operator == (const basic_string < charT, traits, Allocator > & lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template < class charT, class traits, class Allocator >
  bool operator==(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template < class charT, class traits, class Allocator>
  bool operator == (const basic_string < charT, traits, Allocator > & lhs,
                  const charT* rhs);
template < class charT, class traits, class Allocator >
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator!=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template < class charT, class traits, class Allocator >
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template < class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template < class charT, class traits, class Allocator >
  bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator< (const charT* lhs,</pre>
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template < class charT, class traits, class Allocator >
  bool operator> (const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

```
template < class charT, class traits, class Allocator>
 bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template < class charT, class traits, class Allocator >
 bool operator <= (const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template < class charT, class traits, class Allocator >
 bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template < class charT, class traits, class Allocator >
 bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template < class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
// 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);
template<class charT, class traits, class Allocator>
  basic_ostream<charT, traits>&
    operator << (basic_ostream < charT, traits > & os,
               const basic_string<charT,traits,Allocator>& str);
template < class charT, class traits, class Allocator >
 basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>& is,
            basic_string<charT,traits,Allocator>& str,
            charT delim);
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);
template < class charT, class traits, class Allocator >
 basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>&& is,
            basic_string<charT,traits,Allocator>& str);
// basic_string typedef names
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;
```

```
// 21.5, numeric conversions:
int stoi(const string& str, size_t *idx = 0, int base = 10);
long stol(const string& str, size_t *idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t *idx = 0, int base = 10);
long long stoll(const string& str, size_t *idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t *idx = 0, int base = 10);
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(unsigned 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, int base = 10);
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(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);
// 21.6, hash support:
template <class T> struct hash;
template <> struct hash<string>;
template <> struct hash<u16string>;
template <> struct hash<u32string>;
template <> struct hash<wstring>;
```

#### 21.4 Class template basic\_string

[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.

<sup>2</sup> 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. <sup>233</sup>

<sup>3</sup> The iterators supported by basic\_string are random access iterators (24.2.7).

233) Allocator::value\_type must name the same type as charT (21.4.1).

```
In all cases, size() <= capacity().
<sup>5</sup> The functions described in this Clause can report two kinds of errors, each associated with an exception
      - 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).
     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;
                                    const_reference;
         typedef const value_type&
         typedef typename allocator_traits<Allocator>::pointer
                                                                           pointer;
         typedef typename allocator_traits<Allocator>::const_pointer
                                                                           const_pointer;
         typedef implementation-defined
                                                                       // See 23.2
                                                      iterator;
         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) noexcept;
         basic_string(const basic_string& str, size_type pos, size_type n = npos,
                      const Allocator& a = Allocator());
         basic_string(const charT* s,
                      size_type n, const Allocator& a = Allocator());
         basic_string(const charT* s, const Allocator& a = Allocator());
         basic_string(size_type n, charT c, const Allocator& a = Allocator());
         template < class InputIterator>
           basic_string(InputIterator begin, InputIterator end,
                        const Allocator& a = Allocator());
         basic_string(initializer_list<charT>, const Allocator& = Allocator());
         basic_string(const basic_string&, const Allocator&);
         basic_string(basic_string&&, const Allocator&);
        ~basic_string();
         basic_string& operator=(const basic_string& str);
         basic_string& operator=(basic_string&& str) noexcept;
         basic_string& operator=(const charT* s);
         basic_string& operator=(charT c);
         basic_string& operator=(initializer_list<charT>);
         // 21.4.3, iterators:
```

```
iterator
             begin() noexcept;
const_iterator begin() const noexcept;
          end() noexcept;
iterator
const_iterator end() const noexcept;
reverse_iterator
                       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse iterator
                  rend() noexcept;
const_reverse_iterator rend() const noexcept;
                       cbegin() const noexcept;
const_iterator
const_iterator
                       cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;
// 21.4.4, capacity:
size_type size() const noexcept;
size_type length() const noexcept;
size_type max_size() const noexcept;
void resize(size_type n, charT c);
void resize(size_type n);
size_type capacity() const noexcept;
void reserve(size_type res_arg = 0);
void shrink_to_fit();
void clear() noexcept;
bool empty() const noexcept;
// 21.4.5, element access:
const_reference operator[](size_type pos) const;
reference
                operator[](size_type pos);
const_reference at(size_type n) const;
reference
              at(size_type n);
const charT& front() const;
charT& front();
const charT& back() const;
charT& back();
// 21.4.6, modifiers:
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& operator+=(initializer_list<charT>);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
                     size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template < class InputIterator>
 basic_string& append(InputIterator first, InputIterator last);
basic_string& append(initializer_list<charT>);
void push_back(charT c);
basic_string& assign(const basic_string& str);
```

```
basic_string& assign(basic_string&& str) noexcept;
basic_string& assign(const basic_string& str, size_type pos,
                     size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template < class InputIterator>
 basic string& assign(InputIterator first, InputIterator last);
basic_string& assign(initializer_list<charT>);
basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
iterator insert(const_iterator p, size_type n, charT c);
template < class InputIterator>
  iterator insert(const_iterator p, InputIterator first, InputIterator last);
iterator insert(const_iterator p, initializer_list<charT>);
basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator p);
iterator erase(const_iterator first, const_iterator last);
void pop_back();
basic_string& replace(size_type pos1, size_type n1,
                      const basic_string& str);
basic_string& replace(size_type pos1, size_type n1,
                      const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s,
                      size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2,
                      charT c);
basic_string& replace(const_iterator i1, const_iterator i2,
          const basic_string& str);
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s,
                      size_type n);
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);
basic_string& replace(const_iterator i1, const_iterator i2,
                      size_type n, charT c);
template<class InputIterator>
  basic_string& replace(const_iterator i1, const_iterator i2,
                        InputIterator j1, InputIterator j2);
basic_string& replace(const_iterator, const_iterator, initializer_list<charT>);
size_type copy(charT* s, size_type n, size_type pos = 0) const;
void swap(basic_string& str);
// 21.4.7, string operations:
```

const charT\* c\_str() const noexcept;

```
const charT* data() const noexcept;
  allocator_type get_allocator() const noexcept;
  size_type find (const basic_string& str, size_type pos = 0) const noexcept;
  size_type find (const charT* s, size_type pos, size_type n) const;
  size_type find (const charT* s, size_type pos = 0) const;
  size_type find (charT c, size_type pos = 0) const noexcept;
  size_type rfind(const basic_string& str, size_type pos = npos) const noexcept;
  size_type rfind(const charT* s, size_type pos, size_type n) const;
  size_type rfind(const charT* s, size_type pos = npos) const;
  size_type rfind(charT c, size_type pos = npos) const noexcept;
  size_type find_first_of(const basic_string& str,
                          size_type pos = 0) const noexcept;
  size_type find_first_of(const charT* s,
                          size_type pos, size_type n) const;
  size_type find_first_of(const charT* s, size_type pos = 0) const;
  size_type find_first_of(charT c, size_type pos = 0) const noexcept;
  size_type find_last_of (const basic_string& str,
                          size_type pos = npos) const noexcept;
  size_type find_last_of (const charT* s,
                          size_type pos, size_type n) const;
  size_type find_last_of (const charT* s, size_type pos = npos) const;
  size_type find_last_of (charT c, size_type pos = npos) const noexcept;
  size_type find_first_not_of(const basic_string& str,
              size_type pos = 0) const noexcept;
  size_type find_first_not_of(const charT* s, size_type pos,
                              size_type n) const;
  size_type find_first_not_of(const charT* s, size_type pos = 0) const;
  size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
  size_type find_last_not_of (const basic_string& str,
                              size_type pos = npos) const noexcept;
  size_type find_last_not_of (const charT* s, size_type pos,
                              size_type n) const;
  size_type find_last_not_of (const charT* s,
                              size_type pos = npos) const;
  size_type find_last_not_of (charT c, size_type pos = npos) const noexcept;
  basic_string substr(size_type pos = 0, size_type n = npos) const;
  int compare(const basic_string& str) const noexcept;
  int compare(size_type pos1, size_type n1,
              const basic_string& str) const;
  int compare(size_type pos1, size_type n1,
              const basic_string& str,
              size_type pos2, size_type n2) const;
  int compare(const charT* s) const;
  int compare(size_type pos1, size_type n1,
              const charT* s) const;
  int compare(size_type pos1, size_type n1,
              const charT* s, size_type n2) const;
};
```

### 21.4.1 basic\_string general requirements

[string.require]

If any operation would cause size() to exceed max\_size(), that operation shall throw an exception object of type length\_error.

- <sup>2</sup> If any member function or operator of basic\_string throws an exception, that function or operator shall have no other effect.
- No erase() or pop\_back() member function shall throw any exceptions.
- 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.
- 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().
- 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.<sup>234</sup>
  - 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 63.

Element	Value
data()	a non-null pointer that is copyable and can have 0
	added to it
size()	0
capacity()	an unspecified value

Table 63 — basic\_string(const Allocator&) effects

```
basic_string(const basic_string<charT,traits,Allocator>& str);
basic_string(basic_string<charT,traits,Allocator>&& str) noexcept;
```

- 2 Effects: Constructs an object of class basic\_string as indicated in Table 64. In the second form, str is left in a valid state with an unspecified value.
- 3 Throws: The second form throws nothing if the allocator's move constructor throws nothing.

234) For example, as an argument to non-member functions swap() (21.4.8.8), operator>>() (21.4.8.9), and getline() (21.4.8.9), or as an argument to basic\_string::swap()

Table 64 —	basic	string	(const	basic	string&)	effects

Element	Value
data()	points at the first element of an allocated copy
	of the array whose first element is pointed at by
	str.data()
size()	str.size()
capacity()	a value at least as large as size()

- 4 Requires: pos <= str.size()
- 5 Throws: out\_of\_range if pos > str.size().
- Effects: Constructs an object of class basic\_string and determines the effective length rlen of the initial string value as the smaller of n and str.size() pos, as indicated in Table 65.

 $Table \ 65 -- \verb|basic_string| (\verb|const| basic_string| \&|, \ \verb|size_type|, \ \verb|size_type|, \ \verb|const| Allocator| \&|) \ effects$ 

Element	Value
data()	points at the first element of an allocated copy of
	rlen consecutive elements of the string controlled
	by str beginning at position pos
size()	rlen
capacity()	a value at least as large as size()

- 7 Requires: s shall not be a null pointer and n < npos.
- 8 Effects: Constructs an object of class basic\_string and determines its initial string value from the array of charT of length n whose first element is designated by s, as indicated in Table 66.

Table 66 — basic\_string(const charT\*, size\_type, const Allocator&) effects

Element	Value
data()	points at the first element of an allocated copy of
	the array whose first element is pointed at by s
size()	n
capacity()	a value at least as large as size()

basic\_string(const charT\* s, const Allocator& a = Allocator());

- 9 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 67.

Table 67 — basic\_string(const charT\*, const Allocator&) effects

Element	Value
data()	points at the first element of an allocated copy of
	the array whose first element is pointed at by s
size()	traits::length(s)
capacity()	a value at least as large as size()

11 Remarks: Uses traits::length().

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

- 12 Requires: n < npos
- Effects: Constructs an object of class basic\_string and determines its initial string value by repeating the char-like object c for all n elements, as indicated in Table 68.

Table 68 — basic\_string(size\_t, charT, const Allocator&) effects

Element	Value
data()	points at the first element of an allocated array of
	${\tt n}$ elements, each storing the initial value ${\tt c}$
size()	n
capacity()	a value at least as large as size()

- Effects: If InputIterator is an integral type, equivalent to
  basic\_string(static\_cast<size\_type>(begin), static\_cast<value\_type>(end), a)
- Otherwise constructs a string from the values in the range [begin, end), as indicated in the Sequence Requirements table (see 23.2.3).

```
basic_string(initializer_list<charT> i1, const Allocator& a = Allocator());
```

16 Effects: Same as basic\_string(il.begin(), il.end(), a).

```
basic_string(const basic_string& str, const Allocator& alloc);
basic_string(basic_string&& str, const Allocator& alloc);
```

- Effects: Constructs an object of class basic\_string as indicated in Table 69. 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.

Table 69 — basic\_string(const basic\_string&, const Allocator&) and basic\_string(basic\_string&&, const Allocator&) effects

Element	Value
data()	points at the first element of an allocated copy of
	the array whose first element is pointed at by the
	original value of str.data().
size()	the original value of str.size()
capacity()	a value at least as large as size()
<pre>get_allocator()</pre>	alloc

basic\_string<charT,traits,Allocator>&
 operator=(const basic\_string<charT,traits,Allocator>& str);

- 19 Effects: If \*this and str are not the same object, modifies \*this as shown in Table 70.
- If \*this and str are the same object, the member has no effect.
- 21 Returns: \*this

Table 70 — operator=(const basic\_string<charT, traits, Allocator>&) effects

Element	Value
data()	points at the first element of an allocated copy
	of the array whose first element is pointed at by
	str.data()
size()	str.size()
capacity()	a value at least as large as size()

basic\_string<charT,traits,Allocator>&
 operator=(basic\_string<charT,traits,Allocator>&& str) noexcept;

- Effects: If \*this and str are not the same object, modifies \*this as shown in Table 71. [Note: A valid implementation is swap(str). —end note]
- 23 If \*this and str are the same object, the member has no effect.
- 24 Returns: \*this

Table 71 — operator=(const basic\_string<charT, traits, Allocator>&&) effects

Element	Value
data()	points at the array whose first element was pointed
	at by str.data()
size()	previous value of str.size()
capacity()	a value at least as large as size()

```
basic_string<charT,traits,Allocator>&
      operator=(const charT* s);
25
         Returns: *this = basic_string<charT, traits, Allocator>(s).
26
         Remarks: Uses traits::length().
   basic_string<charT,traits,Allocator>& operator=(charT c);
27
         Returns: *this = basic_string<charT, traits, Allocator>(1,c).
   basic_string& operator=(initializer_list<charT> il);
28
         Effects: *this = basic_string(il).
29
         Returns: *this.
                                                                                      [string.iterators]
   21.4.3
            basic string iterator support
   iterator
                   begin() noexcept;
   const_iterator begin() const noexcept;
   const_iterator cbegin() const noexcept;
         Returns: An iterator referring to the first character in the string.
                   end() noexcept;
   const_iterator end() const noexcept;
   const_iterator cend() const noexcept;
2
         Returns: An iterator which is the past-the-end value.
   reverse_iterator
                           rbegin() noexcept;
   const_reverse_iterator rbegin() const noexcept;
   const_reverse_iterator crbegin() const noexcept;
         Returns: An iterator which is semantically equivalent to reverse_iterator(end()).
   reverse_iterator
                           rend() noexcept;
   const_reverse_iterator rend() const noexcept;
   const_reverse_iterator crend() const noexcept;
4
         Returns: An iterator which is semantically equivalent to reverse_iterator(begin()).
   21.4.4 basic_string capacity
                                                                                      [string.capacity]
   size_type size() const noexcept;
1
         Returns: A count of the number of char-like objects currently in the string.
         Complexity: constant time.
   size_type length() const noexcept;
3
         Returns: size().
   § 21.4.4
                                                                                                      619
```

```
size_type max_size() const noexcept;
4
          Returns: The size of the largest possible string.
5
          Complexity: constant time.
    void resize(size_type n, charT c);
6
          Requires: n <= max_size()
7
          Throws: length error if n > max size().
8
          Effects: Alters the length of the string designated by *this as follows:
           — If n <= size(), the function replaces the string designated by *this with a string of length n
              whose elements are a copy of the initial elements of the original string designated by *this.
           — If n > size(), the function replaces the string designated by *this with a string of length n
              whose first size() elements are a copy of the original string designated by *this, and whose
              remaining elements are all initialized to c.
    void resize(size_type n);
9
         Effects: resize(n,charT()).
    size_type capacity() const noexcept;
10
         Returns: The size of the allocated storage in the string.
    void reserve(size_type res_arg=0);
11
         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.
12
         Effects: After reserve(), capacity() is greater or equal to the argument of reserve. [Note: Calling
         reserve() with a res_arg argument less than capacity() is in effect a non-binding shrink request.
         A call with res arg <= size() is in effect a non-binding shrink-to-fit request. — end note
13
          Throws: length_error if res_arg > max_size().235
    void shrink_to_fit();
14
         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 clear() noexcept;
15
         Effects: Behaves as if the function calls:
           erase(begin(), end());
    bool empty() const noexcept;
16
          Returns: size() == 0.
    235) reserve() uses allocator_traits<Allocator>::allocate() which may throw an appropriate exception.
    § 21.4.4
                                                                                                          620
```

```
21.4.5 basic_string element access
                                                                                        [string.access]
   const_reference operator[](size_type pos) const;
   reference
                    operator[](size_type pos);
1
         Requires: pos <= size().
2
         Returns: *(begin() + pos) if pos < size(). Otherwise, returns a reference to an object of type
         charT with value charT(), where modifying the object leads to undefined behavior.
3
         Throws: Nothing.
4
         Complexity: constant time.
   const_reference at(size_type pos) const;
                    at(size_type pos);
   reference
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
                                                                                     [string.modifiers]
   21.4.6.1 basic_string::operator+=
                                                                                         [string::op+=]
   basic_string&
     operator+=(const basic_string& str);
         Effects: Calls append(str).
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 push_back(c);
6
         Returns: *this.
   § 21.4.6.1
                                                                                                     621
```

```
basic_string& operator+=(initializer_list<charT> il);
         Effects: Calls append(i1).
         Returns: *this.
                                                                                          [string::append]
   21.4.6.2 basic string::append
   basic_string&
      append(const basic_string& str);
         Effects: Calls append(str.data(), str.size()).
2
         Returns: *this.
   basic_string&
      append(const basic_string& str, size_type pos, size_type n);
3
         Requires: pos <= str.size()</pre>
4
         Throws: out_of_range if pos > str.size().
5
         Effects: Determines the effective length rlen of the string to append as the smaller of n and str.size()
         - pos and calls append(str.data() + pos, rlen).
6
         Returns: *this.
   basic_string&
      append(const charT* s, size_type n);
7
         Requires: s points to an array of at least n elements of charT.
8
         Throws: length_error if size() + n > max_size().
9
         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.
10
         Returns: *this.
   basic_string& append(const charT* s);
11
         Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
12
         Effects: Calls append(s, traits::length(s)).
13
         Returns: *this.
   basic_string& append(size_type n, charT c);
14
         Effects: Equivalent to append(basic_string(n, c)).
15
         Returns: *this.
   template < class InputIterator>
     basic_string& append(InputIterator first, InputIterator last);
```

§ 21.4.6.2

```
16
         Requires: [first,last) is a valid range.
17
         Effects: Equivalent to append(basic_string(first, last)).
18
         Returns: *this.
    basic_string& append(initializer_list<charT> il);
19
         Effects: Calls append(il.begin(), il.size()).
20
         Returns: *this.
    void push_back(charT c)
21
         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);
1
         Effects: Equivalent to assign(str, 0, npos).
2
         Returns: *this.
    basic_string& assign(basic_string&& str) noexcept;
         Effects: The function replaces the string controlled by *this with a string of length str.size() whose
         elements are a copy of the string controlled by str. [Note: A valid implementation is swap(str).
         end note]
3
         Returns: *this.
    basic_string&
      assign(const basic_string& str, size_type pos,
             size_type n);
4
         Requires: pos <= str.size()
5
         Throws: out_of_range if pos > str.size().
6
         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).
7
         Returns: *this.
    basic_string& assign(const charT* s, size_type n);
8
         Requires: s points to an array of at least n elements of charT.
9
         Throws: length_error if n > max_size().
10
         Effects: Replaces the string controlled by *this with a string of length n whose elements are a copy
         of those pointed to by s.
11
         Returns: *this.
    basic_string& assign(const charT* s);
                                                                                                        623
    § 21.4.6.3
```

```
12
         Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
13
         Effects: Calls assign(s, traits::length(s)).
14
         Returns: *this.
   basic_string& assign(initializer_list<charT> il);
15
         Effects: Calls assign(il.begin(), il.size()).
16
         *this.
   basic_string& assign(size_type n, charT c);
18
         Effects: Equivalent to assign(basic_string(n, c)).
19
         Returns: *this.
   template < class InputIterator>
      basic_string& assign(InputIterator first, InputIterator last);
20
         Effects: Equivalent to assign(basic_string(first, last)).
21
         Returns: *this.
   21.4.6.4 basic_string::insert
                                                                                          [string::insert]
   basic_string&
      insert(size_type pos1,
             const basic_string& str);
1
         Requires: pos <= size().
2
         Throws: out_of_range if pos > size().
3
         Effects: Calls insert(pos, str.data(), str.size()).
4
         Returns: *this.
   basic_string&
      insert(size_type pos1,
             const basic_string& str,
             size_type pos2, size_type n);
5
         Requires: pos1 <= size() and pos2 <= str.size()
6
         Throws: out_of_range if pos1 > size() or pos2 > str.size().
7
         Effects: Determines the effective length rlen of the string to insert as the smaller of n and str.size()
         - pos2 and calls insert(pos1, str.data() + pos2, rlen).
8
         Returns: *this.
   basic_string&
      insert(size_type pos, const charT* s, size_type n);
```

§ 21.4.6.4 624

```
9
          Requires: s points to an array of at least n elements of charT and pos <= size().
10
          Throws: out_of_range if pos > size() or length_error if size() + n > max_size().
11
          Effects: Replaces the string controlled by *this with a string of length size() + n whose first pos
         elements are a copy of the initial elements of the original string controlled by *this and whose next
         n elements are a copy of the elements in s and whose remaining elements are a copy of the remaining
         elements of the original string controlled by *this.
12
         Returns: *this.
    basic_string&
      insert(size_type pos, const charT* s);
13
          Requires: pos <= size() and s points to an array of at least traits::length(s) + 1 elements of
         charT.
14
          Effects: Equivalent to insert(pos, s, traits::length(s)).
15
          Returns: *this.
    basic_string&
      insert(size_type pos, size_type n, charT c);
16
         Effects: Equivalent to insert(pos, basic_string(n, c)).
17
          Returns: *this.
    iterator insert(const_iterator p, charT c);
18
          Requires: p is a valid iterator on *this.
          Effects: inserts a copy of c before the character referred to by p.
19
20
          Returns: An iterator which refers to the copy of the inserted character.
    iterator insert(const_iterator p, size_type n, charT c);
21
          Requires: p is a valid iterator on *this.
22
          Effects: inserts n copies of c before the character referred to by p.
23
          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);
^{24}
          Requires: p is a valid iterator on *this. [first,last) is a valid range.
25
          Effects: Equivalent to insert(p - begin(), basic_string(first, last)).
26
          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);
27
          Effects: insert(p, il.begin(), il.end()).
         Returns: An iterator which refers to the copy of the first inserted character, or p if i1 is empty.
    § 21.4.6.4
                                                                                                          625
```

```
21.4.6.5 basic_string::erase
                                                                                             [string::erase]
   basic_string<charT,traits,Allocator>&
      erase(size_type pos = 0, size_type n = npos);
1
         Requires: pos <= size()
2
         Throws: out_of_range if pos > size().
3
         Effects: Determines the effective length xlen of the string to be removed as the smaller of n and
         size() - pos.
4
         The function then replaces the string controlled by *this with a string of length size() - xlen
         whose first 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.
5
         Returns: *this.
   iterator erase(const_iterator p);
6
         Effects: removes the character referred to by p.
7
         Returns: An iterator which points to the element immediately following p prior to the element being
         erased. If no such element exists, end() is returned.
   iterator erase(const_iterator first, const_iterator last);
8
         Requires: first and last are valid iterators on *this, defining a range [first,last).
9
         Effects: removes the characters in the range [first,last).
10
         Returns: An iterator which points to the element pointed to by last prior to the other elements being
         erased. If no such element exists, end() is returned.
   void pop_back();
11
         Requires: !empty()
12
         Effects: Equivalent to erase(size() - 1, 1).
                                                                                          [string::replace]
   21.4.6.6 basic_string::replace
   basic_string&
      replace(size_type pos1, size_type n1,
              const basic_string& str);
1
         Requires: pos1 <= size().
2
         Throws: out_of_range if pos1 > size().
3
         Effects: Calls replace(pos1, n1, str.data(), str.size()).
4
         Returns: *this.
   basic_string&
     replace(size_type pos1, size_type n1,
              const basic_string& str,
              size_type pos2, size_type n2);
                                                                                                        626
   § 21.4.6.6
```

```
5
         Requires: pos1 <= size() and pos2 <= str.size().
6
         Throws: out_of_range if pos1 > size() or pos2 > str.size().
7
         Effects: Determines the effective length rlen of the string to be inserted as the smaller of n2 and
         str.size() - pos2 and calls replace(pos1, n1, str.data() + pos2, rlen).
         Returns: *this.
   basic_string&
     replace(size_type pos1, size_type n1, const charT* s, size_type n2);
9
         Requires: pos1 <= size() and s points to an array of at least n2 elements of charT.
10
         Throws: out_of_range if pos1 > size() or length_error if the length of the resulting string would
         exceed max_size() (see below).
11
         Effects: Determines the effective length xlen of the string to be removed as the smaller of n1 and
         size() - pos1. If size() - xlen >= max_size() - n2 throws length_error. Otherwise, the func-
         tion replaces the string controlled by *this with a string of length size() - xlen + n2 whose first
         pos1 elements are a copy of the initial elements of the original string controlled by *this, whose next
         n2 elements are a copy of the initial n2 elements of s, and whose remaining elements are a copy of the
         elements of the original string controlled by *this beginning at position pos + xlen.
12
         Returns: *this.
   basic_string&
      replace(size_type pos, size_type n, const charT* s);
13
         Requires: pos <= size() and s points to an array of at least traits::length(s) + 1 elements of
14
         Effects: Calls replace(pos, n, s, traits::length(s)).
15
         Returns: *this.
   basic_string&
      replace(size_type pos1, size_type n1,
              size_type n2, charT c);
16
         Effects: Equivalent to replace(pos1, n1, basic_string(n2, c)).
17
         Returns: *this.
   basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);
18
         Requires: [begin(),i1) and [i1,i2) are valid ranges.
19
         Effects: Calls replace(i1 - begin(), i2 - i1, str).
20
         Returns: *this.
   basic_string&
     replace(const_iterator i1, const_iterator i2, const charT* s, size_type n);
```

§ 21.4.6.6

```
21
         Requires: [begin(),i1) and [i1,i2) are valid ranges and s points to an array of at least n elements
         of charT.
22
         Effects: Calls replace(i1 - begin(), i2 - i1, s, n).
23
         Returns: *this.
   basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);
24
         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.
25
         Effects: Calls replace(i1 - begin(), i2 - i1, s, traits::length(s)).
26
         Returns: *this.
   basic_string& replace(const_iterator i1, const_iterator i2, size_type n,
                          charT c);
27
         Requires: [begin(),i1) and [i1,i2) are valid ranges.
28
         Effects: Calls replace(i1 - begin(), i2 - i1, basic_string(n, c)).
29
         Returns: *this.
   template < class InputIterator>
     basic_string& replace(const_iterator i1, const_iterator i2,
                            InputIterator j1, InputIterator j2);
30
         Requires: [begin(),i1), [i1,i2) and [j1,j2) are valid ranges.
31
         Effects: Calls replace(i1 - begin(), i2 - i1, basic_string(j1, j2)).
32
         Returns: *this.
   basic_string& replace(const_iterator i1, const_iterator i2,
                          initializer_list<charT> il);
33
         Requires: [begin(),i1) and [i1,i2) are valid ranges.
34
         Effects: Calls replace(i1 - begin(), i2 - i1, il.begin(), il.size()).
35
         *this.
                                                                                            [string::copy]
   21.4.6.7 basic_string::copy
   size_type copy(charT* s, size_type n, size_type pos = 0) const;
1
         Requires: pos <= size()
2
         Throws: out_of_range if pos > size().
3
         Effects: Determines the effective length rlen of the string to copy as the smaller of n and size() -
         pos. s shall designate an array of at least rlen elements.
         The function then replaces the string designated by s with a string of length rlen whose elements are
         a copy of the string controlled by *this beginning at position pos.
         The function does not append a null object to the string designated by s.
4
         Returns: rlen.
```

628

§ 21.4.6.7

```
21.4.6.8 basic_string::swap
                                                                                          [string::swap]
   void swap(basic_string<charT,traits,Allocator>& s);
1
        Postcondition: *this contains the same sequence of characters that was in s, s contains the same
        sequence of characters that was in *this.
2
        Throws: Nothing.
3
        Complexity: constant time.
                                                                                           [string.ops]
   21.4.7 basic_string string operations
   21.4.7.1
                                                                                      [string.accessors]
            basic_string accessors
   const charT* c_str() const noexcept;
   const charT* data() const noexcept;
1
        Returns: A pointer p such that p + i == &operator[](i) for each i in [0,size()].
2
        Complexity: constant time.
3
        Requires: The program shall not alter any of the values stored in the character array.
   allocator_type get_allocator() const noexcept;
4
        Returns: A copy of the Allocator object used to construct the string or, if that allocator has been
        replaced, a copy of the most recent replacement.
                                                                                           [string::find]
   21.4.7.2 basic_string::find
   size_type find(const basic_string& str,
                  size_type pos = 0) const noexcept;
        Effects: Determines the lowest position xpos, if possible, such that both of the following conditions
        obtain:
         — pos <= xpos and xpos + str.size() <= size();</pre>
         — traits::eq(at(xpos+I), str.at(I)) for all elements I of the string controlled by str.
2
        Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.
3
        Remarks: Uses traits::eq().
   size_type find(const charT* s, size_type pos, size_type n) const;
4
        Returns: find(basic_string<charT,traits,Allocator>(s,n),pos).
   size_type find(const charT* s, size_type pos = 0) const noexcept;
5
        Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
6
        Returns: find(basic_string(s), pos).
   size_type find(charT c, size_type pos = 0) const noexcept;
        Returns: find(basic_string<charT,traits,Allocator>(1,c), pos).
   § 21.4.7.2
                                                                                                     629
```

```
[string::rfind]
   21.4.7.3 basic_string::rfind
   size_type rfind(const basic_string& str,
                   size_type pos = npos) const noexcept;
1
        Effects: Determines the highest position xpos, if possible, such that both of the following conditions
        obtain:
          — xpos <= pos and xpos + str.size() <= size();</pre>
         — traits::eq(at(xpos+I), str.at(I)) for all elements I of the string controlled by str.
2
        Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.
3
        Remarks: Uses traits::eq().
   size_type rfind(const charT* s, size_type pos, size_type n) const;
4
        Returns: rfind(basic_string(s, n), pos).
   size_type rfind(const charT* s, size_type pos = npos) const;
5
        Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
6
        Returns: rfind(basic_string(s), pos).
   size_type rfind(charT c, size_type pos = npos) const noexcept;
        Returns: rfind(basic_string<charT,traits,Allocator>(1,c),pos).
                                                                                   [string::find.first.of]
   21.4.7.4 basic_string::find_first_of
   size_type
     find_first_of(const basic_string& str,
                   size_type pos = 0) const noexcept;
        Effects: Determines the lowest position xpos, if possible, such that both of the following conditions
        obtain:
         — pos <= xpos and xpos < size();</pre>
         — traits::eq(at(xpos), str.at(I)) for some element I of the string controlled by str.
2
        Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.
3
        Remarks: Uses traits::eq().
   size_type
     find_first_of(const charT* s, size_type pos, size_type n) const;
        Returns: find_first_of(basic_string(s, n), pos).
   size_type find_first_of(const charT* s, size_type pos = 0) const;
5
        Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
6
        Returns: find_first_of(basic_string(s), pos).
   size_type find_first_of(charT c, size_type pos = 0) const noexcept;
7
        Returns: find_first_of(basic_string<charT,traits,Allocator>(1,c), pos).
   § 21.4.7.4
                                                                                                     630
```

```
[string::find.last.of]
   21.4.7.5 basic_string::find_last_of
   size_type
     find_last_of(const basic_string& str,
                  size_type pos = npos) const noexcept;
        Effects: Determines the highest position xpos, if possible, such that both of the following conditions
        obtain:
         — xpos <= pos and xpos < size();</pre>
         — traits::eq(at(xpos), str.at(I)) for some element I of the string controlled by str.
2
        Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.
3
        Remarks: Uses traits::eq().
   size_type find_last_of(const charT* s, size_type pos, size_type n) const;
4
        Returns: find last of (basic string(s, n), pos).
   size_type find_last_of(const charT* s, size_type pos = npos) const;
5
        Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
6
        Returns: find_last_of(basic_string(s), pos).
   size_type find_last_of(charT c, size_type pos = npos) const noexcept;
        Returns: find_last_of(basic_string<charT, traits, Allocator>(1,c),pos).
   21.4.7.6 basic_string::find_first_not_of
                                                                               [string::find.first.not.of]
   size_type
     find_first_not_of(const basic_string& str,
                       size_type pos = 0) const noexcept;
        Effects: Determines the lowest position xpos, if possible, such that both of the following conditions
        obtain:
         — pos <= xpos and xpos < size();</pre>
         — traits::eq(at(xpos), str.at(I)) for no element I of the string controlled by str.
2
        Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.
3
        Remarks: Uses traits::eq().
   size_type
     find_first_not_of(const charT* s, size_type pos, size_type n) const;
        Returns: find_first_not_of(basic_string(s, n), pos).
   size_type find_first_not_of(const charT* s, size_type pos = 0) const;
```

§ 21.4.7.6

```
Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
6
        Returns: find_first_not_of(basic_string(s), pos).
   size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
7
        Returns: find_first_not_of(basic_string(1, c), pos).
   21.4.7.7 basic_string::find_last_not_of
                                                                               [string::find.last.not.of]
   size_type
     find_last_not_of(const basic_string& str,
                      size_type pos = npos) const noexcept;
1
        Effects: Determines the highest position xpos, if possible, such that both of the following conditions
        obtain:
         — xpos <= pos and xpos < size();</pre>
         — traits::eq(at(xpos), str.at(I)) for no element I of the string controlled by str.
2
        Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.
3
        Remarks: Uses traits::eq().
   size_type find_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;
5
        Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
6
        Returns: find_last_not_of(basic_string(s), pos).
   size_type find_last_not_of(charT c, size_type pos = npos) const noexcept;
        Returns: find_last_not_of(basic_string(1, c), pos).
   21.4.7.8 basic_string::substr
                                                                                        [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).
```

§ 21.4.7.8

#### 21.4.7.9 basic\_string::compare

1

[string::compare]

int compare(const basic\_string& str) const noexcept;

Effects: Determines the effective length rlen of the strings to compare as the smallest of size() and str.size(). The function then compares the two strings by calling traits::compare(data(), str.data(), rlen).

2 Returns: The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 72.

Table 72	- compare()	results
----------	-------------	---------

Condition	Return Value
size() < str.size()	< 0
size() == str.size()	0
size() > str.size()	> 0

```
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;
4
        Returns: basic_string(*this, pos1, n1).compare(basic_string(str, pos2, n2)).
  int compare(const charT *s) const;
5
        Returns: compare(basic_string(s)).
  int compare(size_type pos, size_type n1,
              const charT *s) const;
6
        Returns: basic_string(*this, pos, n1).compare(basic_string(s)).
  int compare(size_type pos, size_type n1,
              const charT *s, size_type n2) const;
        Returns: basic_string(*this, pos, n1).compare(basic_string(s, n2)).
                                                                            [string.nonmembers]
  21.4.8 basic_string non-member functions
                                                                                       [string::op+]
  21.4.8.1
             operator+
  template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
      operator+(const basic_string<charT,traits,Allocator>& lhs,
                const basic_string<charT,traits,Allocator>& rhs);
```

§ 21.4.8.1

```
Returns: basic_string<charT, traits, Allocator>(lhs).append(rhs)
   template < class charT, class traits, class Allocator >
     basic_string<charT,traits,Allocator>
       operator+(basic_string<charT,traits,Allocator>&& lhs,
                 const basic_string<charT,traits,Allocator>& rhs);
2
        Returns: std::move(lhs.append(rhs))
   template < class charT, class traits, class Allocator >
     basic_string<charT,traits,Allocator>
       operator+(const basic_string<charT,traits,Allocator>& lhs,
                 basic_string<charT,traits,Allocator>&& rhs);
3
        Returns: std::move(rhs.insert(0, lhs))
   template < class charT, class traits, class Allocator >
     basic_string<charT,traits,Allocator>
       operator+(basic_string<charT,traits,Allocator>&& lhs,
                 basic_string<charT,traits,Allocator>&& rhs);
        Returns: std::move(lhs.append(rhs)) [Note: Or equivalently std::move(rhs.insert(0, lhs))
        -end note
   template < class charT, class traits, class Allocator >
     basic_string<charT,traits,Allocator>
       operator+(const charT* lhs,
                 const basic_string<charT,traits,Allocator>& rhs);
5
        Returns: basic_string<charT,traits,Allocator>(lhs) + rhs.
6
        Remarks: Uses traits::length().
   template < class charT, class traits, class Allocator>
     basic_string<charT,traits,Allocator>
       operator+(const charT* lhs,
                 basic_string<charT,traits,Allocator>&& rhs);
7
        Returns: std::move(rhs.insert(0, lhs)).
8
        Remarks: Uses traits::length().
   template < class charT, class traits, class Allocator >
     basic_string<charT,traits,Allocator>
       operator+(charT lhs,
                 const basic_string<charT,traits,Allocator>& rhs);
9
        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);
                                                                                                    634
   § 21.4.8.1
```

```
10
         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);
11
         Returns: lhs + basic_string<charT,traits,Allocator>(rhs).
12
         Remarks: Uses traits::length().
   template < class charT, class traits, class Allocator >
      basic_string<charT,traits,Allocator>
        operator+(basic_string<charT,traits,Allocator>&& lhs,
                  const charT* rhs);
13
         Returns: std::move(lhs.append(rhs)).
14
         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);
15
         Returns: lhs + basic_string<charT,traits,Allocator>(1,rhs).
   template < class charT, class traits, class Allocator >
     basic_string<charT,traits,Allocator>
       operator+(basic_string<charT,traits,Allocator>&& lhs,
                  charT rhs);
16
         Returns: std::move(lhs.append(1, rhs)).
   21.4.8.2
             operator==
                                                                                   [string::operator==]
   template < class charT, class traits, class Allocator >
     bool operator == (const basic_string < charT, traits, Allocator > & lhs,
                      const basic_string<charT,traits,Allocator>& rhs) noexcept;
         Returns: lhs.compare(rhs) == 0.
   template < class charT, class traits, class Allocator >
     bool operator==(const charT* lhs,
                      const basic_string<charT,traits,Allocator>& rhs) noexcept;
         Returns: rhs == lhs.
   template < class charT, class traits, class Allocator >
      bool operator == (const basic_string < charT, traits, Allocator > & lhs,
                      const charT* rhs) noexcept;
3
         Requires: rhs points to an array of at least traits::length(rhs) + 1 elements of charT.
         Returns: lhs.compare(rhs) == 0.
```

§ 21.4.8.2

```
21.4.8.3 operator!=
                                                                                          [string::op!=]
   template < class charT, class traits, class Allocator >
     bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                     const basic_string<charT,traits,Allocator>& rhs) noexcept;
        Returns: !(lhs == rhs).
   template < class charT, class traits, class Allocator >
     bool operator!=(const charT* lhs,
                     const basic_string<charT,traits,Allocator>& rhs) noexcept;
        Returns: rhs != lhs.
   template < class charT, class traits, class Allocator >
     bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                     const charT* rhs) noexcept;
3
        Requires: rhs points to an array of at least traits::length(rhs) + 1 elements of charT.
4
        Returns: lhs.compare(rhs) != 0.
                                                                                           [string::op<]
   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) noexcept;
        Returns: lhs.compare(rhs) < 0.
   template<class charT, class traits, class Allocator>
     bool operator< (const charT* lhs,</pre>
                     const basic_string<charT,traits,Allocator>& rhs) noexcept;
        Returns: rhs.compare(lhs) > 0.
   template < class charT, class traits, class Allocator >
     bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                     const charT* rhs) noexcept;
        Returns: lhs.compare(rhs) < 0.
                                                                                           [string::op>]
   21.4.8.5
             operator>
   template < class charT, class traits, class Allocator >
     bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                     const basic_string<charT,traits,Allocator>& rhs) noexcept;
        Returns: lhs.compare(rhs) > 0.
   template < class charT, class traits, class Allocator >
     bool operator> (const charT* lhs,
                     const basic_string<charT,traits,Allocator>& rhs) noexcept;
        Returns: rhs.compare(lhs) < 0.
   § 21.4.8.5
                                                                                                     636
```

```
template < class charT, class traits, class Allocator >
 bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs) noexcept;
     Returns: lhs.compare(rhs) > 0.
21.4.8.6
          operator<=
                                                                                     [string::op <=]
template < class charT, class traits, class Allocator >
 bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
     Returns: lhs.compare(rhs) <= 0.
template<class charT, class traits, class Allocator>
 bool operator<=(const charT* lhs,</pre>
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
     Returns: rhs.compare(lhs) >= 0.
template<class charT, class traits, class Allocator>
 bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs) noexcept;
     Returns: lhs.compare(rhs) <= 0.
                                                                                     [string::op>=]
21.4.8.7 operator>=
template < class charT, class traits, class Allocator >
 bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
     Returns: lhs.compare(rhs) >= 0.
template < class charT, class traits, class Allocator >
 bool operator>=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs) noexcept;
     Returns: rhs.compare(lhs) <= 0.
template < class charT, class traits, class Allocator >
 bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs) noexcept;
     Returns: lhs.compare(rhs) >= 0.
21.4.8.8 swap
                                                                                     [string.special]
template < class charT, class traits, class Allocator >
 void swap(basic_string<charT,traits,Allocator>& lhs,
   basic_string<charT,traits,Allocator>& rhs) noexcept;
     Effects: lhs.swap(rhs);
```

§ 21.4.8.8 637

#### 21.4.8.9 Inserters and extractors

[string.io]

Effects: Behaves as a formatted input function (27.7.2.2.1). After constructing a sentry object, if the sentry converts to true, calls str.erase() and then extracts characters from is and appends them to str as if by calling str.append(1,c). If is.width() is greater than zero, the maximum number n of characters appended is is.width(); otherwise n is str.max\_size(). Characters are extracted and appended until any of the following occurs:

- n characters are stored;
- end-of-file occurs on the input sequence;
- isspace(c,is.getloc()) is true for the next available input character c.
- After the last character (if any) is extracted, is.width(0) is called and the sentry object k is destroyed.
- If the function extracts no characters, it calls is.setstate(ios::failbit), which may throw ios\_base::failure (27.5.5.4).
- 4 Returns: is

Effects: Behaves as a formatted output function (27.7.3.6.1). After constructing a sentry object, if this object returns true when converted to a value of type bool, determines padding as described in 22.4.2.2.2, then inserts the resulting sequence of characters seq as if by calling os.rdbuf()->sputn(seq, n), where n is the larger of os.width() and str.size(); then calls os.width(0).

6 Returns: os

5

7

Effects: Behaves as an unformatted input function (27.7.2.3), except that it does not affect the value returned by subsequent calls to basic\_istream<>::gcount(). After constructing a sentry object, if the sentry converts to true, calls str.erase() and then extracts characters from is and appends them to str as if by calling str.append(1, c) until any of the following occurs:

— end-of-file occurs on the input sequence (in which case, the getline function calls is.setstate(ios\_base::eofbit)).

§ 21.4.8.9

```
— traits::eq(c, delim) for the next available input character c (in which case, c is extracted but not appended) (27.5.5.4)
```

- str.max\_size() characters are stored (in which case, the function calls is.setstate(ios\_base ::failbit)) (27.5.5.4)
- The conditions are tested in the order shown. In any case, after the last character is extracted, the sentry object k is destroyed.
- If the function extracts no characters, it calls is.setstate(ios\_base::failbit) which may throw ios\_base::failure (27.5.5.4).
- 10 Returns: is.

1

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.

- 2 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.

5 Returns: The converted result.

Throws: invalid\_argument if strtod or strtold reports that no conversion could be performed. Throws out\_of\_range if strtod or strtold sets errno to ERANGE.

```
string to_string(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", "%llu", "%llu", "%f", or "%Lf", respectively, where buf designates an internal character buffer of sufficient size.

```
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.
- 9 Returns: The converted result.
- Throws: invalid\_argument if wcstol, wcstoll, or wcstoll reports that no conversion could be performed. Throws out\_of\_range if the converted value is outside the range of representable values for the return type.

```
float stof(const wstring& str, size_t *idx = 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.
- 12 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.

```
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"%lu", L"%llu", L"%llu", L"%f", t"%f", or L"%Lf", respectively, where buf designates an internal character buffer of sufficient size buffsz.

## 21.6 Hash support

14

[basic.string.hash]

```
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 (20.8.12).

### 21.7 Null-terminated sequence utilities

[c.strings]

- Tables 74, 75, 76, 77, 78, and 79 describe headers <cctype>, <cwctype>, <cstring>, <cwchar>, <cstdlib> (character conversions), and <cuchar>, respectively.
- The contents of these headers shall be the same as the Standard C Library headers <ctype.h>, <wctype.h>, <string.h>, <wchar.h>, and <stdlib.h> and the C Unicode TR header <uchar.h>, respectively, with the following modifications:
- <sup>3</sup> The headers shall not define the types char16\_t, char32\_t, and wchar\_t (2.12).
- <sup>4</sup> The function signature strchr(const char\*, int) shall be replaced by the two declarations:

both of which shall have the same behavior as the original declaration.

<sup>5</sup> The function signature strpbrk(const char\*, const char\*) shall be replaced by the two declarations:

both of which shall have the same behavior as the original declaration.

<sup>6</sup> The function signature strrchr(const char\*, int) shall be replaced by the two declarations:

both of which shall have the same behavior as the original declaration.

7 The function signature strstr(const char\*, const char\*) shall be replaced by the two declarations:

both of which shall have the same behavior as the original declaration.

8 The function signature memchr(const void\*, int, size\_t) shall be replaced by the two declarations:

```
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.

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.

The function signature wcspbrk(const wchar\_t\*, const wchar\_t\*) shall be replaced by the two declarations:

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:

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:

both of which shall have the same behavior as the original declaration.

The function signature wmemchr(const wwchar\_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.

- <sup>4</sup> The functions strerror and strtok are not required to avoid data races (17.6.5.9).
- Calling the functions listed in Table 73 with an mbstate\_t\* argument of NULL may introduce a data race (17.6.5.9) with other calls to these functions with an mbstate\_t\* argument of NULL.

Table 73 — Potential mbstate\_t data races

mbrlen	mbrtowc	mbsrtowc	mbtowc	wcrtomb
wcsrtomb	wctomb			

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 74 — Header <cctype> synopsis

Type		N	Iame(s)	
Functions:				
isalnum	isblank	isdigit	isprint	isupper
tolower	isalpha	isgraph	ispunct	isxdigit
toupper	iscntrl	islower	isspace	

 $\odot$  ISO/IEC N3337

Table 75 — Header <cwctype> synopsis

Type		N	ame(s)	
Macro:	WEOF			
Types:	wctrans_t	wctype_t	wint_t	
Functions:				
iswalnum	iswctype	iswprint	iswxdigit	wctrans
iswalpha	iswdigit	iswpunct	towctrans	wctype
iswblank	iswgraph	iswspace	towlower	
iswcntrl	iswlower	iswupper	towupper	

Table 76 — Header <cstring> synopsis

Type		N	ame(s)	
Macro:	NULL <cstri< th=""><th colspan="3">NULL <cstring></cstring></th></cstri<>	NULL <cstring></cstring>		
Type:	size_t <cst< th=""><th>ring&gt;</th><th></th><th></th></cst<>	ring>		
Functions:				
memchr	strcat	strcspn	${\tt strncpy}$	strtok
memcmp	strchr	strerror	strpbrk	strxfrm
memcpy	strcmp	strlen	strrchr	
memmove	strcoll	strncat	strspn	
memset	strcpy	$\operatorname{\mathtt{strncmp}}$	strstr	

Table 77 — Header <cwchar> synopsis

Type		Na	ame(s)	
Macros:	NULL	WCHAR_MAX	WCHAR_MIN	WEOF
Types:	mbstate_t	wint_t	size_t	tm
Functions:				
btowc	mbsinit	vwscanf	wcsncpy	wcstoull
fgetwc	mbsrtowcs	wcrtomb	wcspbrk	wcstoul
fgetws	putwchar	wcscat	wcsrchr	wcsxfrm
fputwc	putwc	wcschr	wcsrtombs	wctob
fputws	swprintf	wcscmp	wcsspn	wmemchr
fwide	swscanf	wcscoll	wcsstr	wmemcmp
fwprintf	ungetwc	wcscpy	wcstod	wmemcpy
fwscanf	vfwprintf	wcscspn	wcstof	wmemmove
getwchar	vfwscanf	wcsftime	wcstok	wmemset
getwc	vswprintf	wcslen	wcstold	wprintf
mbrlen	vswscanf	wcsncat	wcstoll	wscanf
mbrtowc	vwprintf	wcsncmp	wcstol	

 $\odot$  ISO/IEC N3337

Table 78 — Header <cstdlib> synopsis

Type		Na	ame(s)	
Macros:	MB_CUR_MAX			
Functions:				
atof	mblen	strtof	strtoul	
atoi	mbtowc	strtol	strtoull	
atol	mbstowcs	strtold	wctomb	
atoll	strtod	strtoll	wcstombs	

Table 79 — Header <cuchar> synopsis

Type		Name(s)
Macros:	STDC_UTF_	16
	STDC_UTF_	32
Functions:	mbrtoc16	c16rtomb
	mbrtoc32	c32rtomb

# 22 Localization library

22.6

# [localization]

#### 22.1 General

## [localization.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 80.

	Subclause	Header(s)
22.3	Locales	<locale></locale>
22.4	Standard locale Categories	
22.5	Standard code conversion facets	<codecvt></codecvt>

<clocale>

Table 80 — Localization library summary

## 22.2 Header <locale> synopsis

[locale.syn]

```
namespace std {
  // 22.3.1, locale:
  class locale;
  template <class Facet> const Facet& use_facet(const locale&);
  template <class Facet> bool
                                      has_facet(const locale&) noexcept;
  // 22.3.3, convenience interfaces:
  template <class charT> bool isspace (charT c, const locale& loc);
  template <class charT> bool isprint (charT c, const locale& loc);
  template <class charT> bool iscntrl (charT c, const locale& loc);
  template <class charT> bool isupper (charT c, const locale& loc);
  template <class charT> bool islower (charT c, const locale& loc);
  template <class charT> bool isalpha (charT c, const locale& loc);
  template <class charT> bool isdigit (charT c, const locale& loc);
  template <class charT> bool ispunct (charT c, const locale& loc);
  template <class charT> bool isxdigit(charT c, const locale& loc);
  template <class charT> bool isalnum (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,</pre>
    class Wide_alloc = std::allocator<Elem>,
    class Byte_alloc = std::allocator<char> > class wstring_convert;
  template <class Codecvt, class Elem = wchar_t,</pre>
     class Tr = char_traits<Elem>> class wbuffer_convert;
  // 22.4.1, ctype:
  class ctype_base;
```

C library locales

§ 22.2 645

```
template <class charT> class ctype;
                                                            // specialization
    template <>
                           class ctype<char>;
    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, 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.<sup>236</sup>
22.3 Locales
                                                                                             [locales]
22.3.1 Class locale
                                                                                              [locale]
  namespace std {
    class locale {
    public:
      // types:
      class facet;
      class id;
      typedef int category;
```

236) In this subclause, the type name  ${\tt struct}$   ${\tt tm}$  is an incomplete type that is defined in  ${\tt <ctime>}$ .

§ 22.3.1

```
static const category // values assigned here are for exposition only
                    = 0.
           none
           collate = 0x010, ctype
                                       = 0x020,
           monetary = 0x040, numeric = 0x080,
                    = 0x100, messages = 0x200,
           all = collate | ctype | monetary | numeric | time | messages;
         // construct/copy/destroy:
         locale() noexcept;
         locale(const locale& other) noexcept;
         explicit locale(const char* std_name);
         explicit locale(const string& std_name);
         locale(const locale& other, const char* std_name, category);
         locale(const locale& other, const string& std_name, category);
         template <class Facet> locale(const locale& other, Facet* f);
         locale(const locale& other, const locale& one, category);
                                      // not virtual
         ~locale();
         const locale& operator=(const locale& other) noexcept;
         template <class Facet> locale combine(const locale& other) const;
         // locale operations:
         basic_string<char>
                                               name() const;
         bool operator==(const locale& other) const;
         bool operator!=(const locale& other) const;
         template <class charT, class traits, class Allocator>
           bool operator()(const basic_string<charT,traits,Allocator>& s1,
                            const basic_string<charT,traits,Allocator>& s2) const;
         // global locale objects:
                      locale global(const locale&);
         static const locale& classic();
       };
     }
<sup>1</sup> Class locale implements a type-safe polymorphic set of facets, indexed by facet type. In other words, a
   facet has a dual role: in one sense, it's just a class interface; at the same time, it's an index into a locale's
   set of facets.
```

Access to the facets of a locale is via two function templates, use\_facet<> and has\_facet<>.

3 [Example: An iostream operator<< might be implemented as: 237

```
template <class charT, class traits>
basic_ostream<charT,traits>&
operator<< (basic_ostream<charT,traits>& s, Date d) {
 typename basic_ostream<charT,traits>::sentry cerberos(s);
 if (cerberos) {
   ios_base::iostate err = ios_base::iostate::goodbit;
    tm tmbuf; d.extract(tmbuf);
   use_facet< time_put<charT,ostreambuf_iterator<charT,traits> > >(
      s.getloc()).put(s, s, s.fill(), err, &tmbuf, 'x');
    s.setstate(err);
                                // might throw
 return s;
```

237) Note that in the call to put the stream is implicitly converted to an ostreambuf\_iterator<charT,traits>.

§ 22.3.1 647

}

— end example]

In the call to use\_facet<Facet>(loc), the type argument chooses a facet, making available all members of the named type. If Facet is not present in a locale, it throws the standard exception bad\_cast. A C++ program can check if a locale implements a particular facet with the function template has\_facet<Facet>(). User-defined facets may be installed in a locale, and used identically as may standard facets (22.4.8).

- <sup>5</sup> [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 (27.7.2.2).) 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.
- <sup>7</sup> 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 (17.6.5.9).

## 22.3.1.1 locale types

[locale.types]

# 22.3.1.1.1 Type locale::category

[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  $(X \mid Y)$ , where X and Y each represent a single category, represents the union of the two categories.

- <sup>2</sup> locale member functions expecting a category argument require one of the category values defined above, or the union of two or more such values. Such a category value identifies a set of locale categories. Each locale category, in turn, identifies a set of locale facets, including at least those shown in Table 81.
- For any locale loc either constructed, or returned by locale::classic(), and any facet Facet shown in Table 81, has\_facet<Facet>(loc) is true. Each locale member function which takes a locale::category argument operates on the corresponding set of facets.
- <sup>4</sup> An implementation is required to provide those specializations for facet templates identified as members of a category, and for those shown in Table 82.
- The provided implementation of members of facets num\_get<charT> and num\_put<charT> calls use\_facet<F> (1) only for facet F of types numpunct<charT> and ctype<charT>, and for locale 1 the value obtained by calling member getloc() on the ios\_base& argument to these functions.

§ 22.3.1.1.1 648

 $\odot$  ISO/IEC N3337

Table 81 — Locale category facets

Category	Includes facets
collate	<pre>collate<char>, collate<wchar_t></wchar_t></char></pre>
ctype	ctype <char>, ctype<wchar_t></wchar_t></char>
	<pre>codecvt<char,char,mbstate_t></char,char,mbstate_t></pre>
	<pre>codecvt<char16_t,char,mbstate_t></char16_t,char,mbstate_t></pre>
	<pre>codecvt<char32_t,char,mbstate_t></char32_t,char,mbstate_t></pre>
	<pre>codecvt<wchar_t,char,mbstate_t></wchar_t,char,mbstate_t></pre>
monetary	<pre>moneypunct<char>, moneypunct<wchar_t></wchar_t></char></pre>
	<pre>moneypunct<char,true>, moneypunct<wchar_t,true></wchar_t,true></char,true></pre>
	<pre>money_get<char>, money_get<wchar_t></wchar_t></char></pre>
	<pre>money_put<char>, money_put<wchar_t></wchar_t></char></pre>
numeric	<pre>numpunct<char>, numpunct<wchar_t></wchar_t></char></pre>
	<pre>num_get<char>, num_get<wchar_t></wchar_t></char></pre>
	<pre>num_put<char>, num_put<wchar_t></wchar_t></char></pre>
time	time_get <char>, time_get<wchar_t></wchar_t></char>
	<pre>time_put<char>, time_put<wchar_t></wchar_t></char></pre>
messages	messages <char>, messages<wchar_t></wchar_t></char>

Table 82 — Required specializations

Category	Includes facets
collate	collate_byname <char>, collate_byname<wchar_t></wchar_t></char>
ctype	ctype_byname <char>, ctype_byname<wchar_t></wchar_t></char>
	<pre>codecvt_byname<char,char,mbstate_t></char,char,mbstate_t></pre>
	<pre>codecvt_byname<char16_t,char,mbstate_t></char16_t,char,mbstate_t></pre>
	<pre>codecvt_byname<char32_t,char,mbstate_t></char32_t,char,mbstate_t></pre>
	<pre>codecvt_byname<wchar_t,char,mbstate_t></wchar_t,char,mbstate_t></pre>
monetary	moneypunct_byname <char,international></char,international>
	<pre>moneypunct_byname<wchar_t,international></wchar_t,international></pre>
	<pre>money_get<c,inputiterator></c,inputiterator></pre>
	<pre>money_put<c,outputiterator></c,outputiterator></pre>
numeric	<pre>numpunct_byname<char>, numpunct_byname<wchar_t></wchar_t></char></pre>
	<pre>num_get<c,inputiterator>, num_put<c,outputiterator></c,outputiterator></c,inputiterator></pre>
time	time_get <char,inputiterator></char,inputiterator>
	time_get_byname <char,inputiterator></char,inputiterator>
	time_get <wchar_t,inputiterator></wchar_t,inputiterator>
	<pre>time_get_byname<wchar_t,inputiterator></wchar_t,inputiterator></pre>
	time_put <char,outputiterator></char,outputiterator>
	<pre>time_put_byname<char,outputiterator></char,outputiterator></pre>
	<pre>time_put<wchar_t,outputiterator></wchar_t,outputiterator></pre>
	<pre>time_put_byname<wchar_t,outputiterator></wchar_t,outputiterator></pre>
messages	<pre>messages_byname<char>, messages_byname<wchar_t></wchar_t></char></pre>

§ 22.3.1.1.1 649

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.

```
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 Class locale::facet

}; }

- 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.
- 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.
- Constructors of all facets defined in this Clause take such an argument and pass it along to their facet base class constructor. All one-argument constructors defined in this Clause are explicit, preventing their participation in automatic conversions.
- For some standard facets a standard "...\_byname" class, derived from it, implements the virtual function semantics equivalent to that facet of the locale constructed by locale(const char\*) with the same name. Each such facet provides a constructor that takes a const char\* argument, which names the locale, and a refs argument, which is passed to the base class constructor. Each such facet also provides a constructor that takes a string argument str and a refs argument, which has the same effect as calling the first constructor with the two arguments str.c\_str() and refs. If there is no "...\_byname" version of a facet, the base class implements named locale semantics itself by reference to other facets.

#### 22.3.1.1.3 Class locale::id

[locale.id]

[locale.facet]

```
namespace std {
  class locale::id {
  public:
    id();
    void operator=(const id&) = delete;
    id(const id&) = delete;
};
}
```

- The class locale::id provides identification of a locale facet interface, used as an index for lookup and to encapsulate initialization.
- <sup>2</sup> [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

§ 22.3.1.1.3

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.cons]
    locale() noexcept;
1
         Default constructor: a snapshot of the current global locale.
2
         Effects: Constructs a copy of the argument last passed to locale::global(locale&), if it has been
         called; else, the resulting facets have virtual function semantics identical to those of locale::classic().
         Note: This constructor is commonly used as the default value for arguments of functions that take a
         const locale& argument. — end note
    locale(const locale& other) noexcept;
3
         Effects: Constructs a locale which is a copy of other.
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 imple-
         ments semantics defined to be associated with that name.
         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);
         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).
```

- Effects: Constructs a locale incorporating all facets from the first argument except that of type Facet, and installs the second argument as the remaining facet. If f is null, the resulting object is a copy of other.
- 15 Remarks: The resulting locale has no name.

template <class Facet> locale(const locale& other, Facet\* f);

§ 22.3.1.2 651

```
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.

17 Remarks: The resulting locale has a name if and only if the first two arguments have names.

```
const locale& operator=(const locale& other) noexcept;
~locale();
```

A non-virtual destructor that throws no exceptions.

#### 22.3.1.3 locale members

18

1

1

[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.
- 2 Returns: The newly created locale.
- 3 Throws: runtime\_error if has\_facet<Facet>(other) is false.
- 4 Remarks: The resulting locale has no name.

```
basic_string<char> name() const;
```

Returns: The name of \*this, if it has one; otherwise, the string "\*". If \*this has a name, then locale(name().c\_str()) is equivalent to \*this. Details of the contents of the resulting string are otherwise implementation-defined.

## 22.3.1.4 locale operators

[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).

- 3 Effects: Compares two strings according to the collate<charT> facet.
- 4 Remarks: This member operator template (and therefore locale itself) satisfies requirements for a comparator predicate template argument (Clause 25) applied to strings.
- 5 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;</pre>
```

§ 22.3.1.4 652

```
6
        Example: A vector of strings v can be collated according to collation rules in locale loc simply
        by (25.4.1, 23.3.6):
          std::sort(v.begin(), v.end(), loc);
         — end example]
   22.3.1.5 locale static members
                                                                                          [locale.statics]
   static locale global(const locale& loc);
1
        Sets the global locale to its argument.
2
        Effects: Causes future calls to the constructor locale() to return a copy of the argument. If the
        argument has a name, does
          std::setlocale(LC_ALL, loc.name().c_str());
        otherwise, the effect on the C locale, if any, is implementation-defined. No library function other
        than locale::global() shall affect the value returned by locale(). [Note: See 22.6 for data race
        considerations when setlocale is invoked. — end note
3
        Returns: The previous value of locale().
   static const locale& classic();
4
        The "C" locale.
5
        Returns: A locale that implements the classic "C" locale semantics, equivalent to the value locale ("C").
6
        Remarks: This locale, its facets, and their member functions, do not change with time.
   22.3.2
            locale globals
                                                                             [locale.global.templates]
   template <class Facet> const Facet& use_facet(const locale& loc);
        Requires: Facet is a facet class whose definition contains the public static member id as defined
        in 22.3.1.1.2.
2
        Returns: A reference to the corresponding facet of loc, if present.
3
        Throws: bad_cast if has_facet<Facet>(loc) is false.
4
        Remarks: The reference returned remains valid at least as long as any copy of loc exists.
   template <class Facet> bool has_facet(const locale& loc) noexcept;
5
        Returns: True if the facet requested is present in loc; otherwise false.
   22.3.3
            Convenience interfaces
                                                                                  [locale.convenience]
   22.3.3.1
             Character classification
                                                                                          [classification]
   template <class charT> bool isspace (charT c, const locale& loc);
   template <class charT> bool isprint (charT c, const locale& loc);
   template <class charT> bool iscntrl (charT c, const locale& loc);
   template <class charT> bool isupper (charT c, const locale& loc);
   template <class charT> bool islower (charT c, const locale& loc);
   template <class charT> bool isalpha (charT c, const locale& loc);
   template <class charT> bool isdigit (charT c, const locale& loc);
```

653

§ 22.3.3.1

```
template <class charT> bool ispunct (charT c, const locale& loc);
   template <class charT> bool isxdigit(charT c, const locale& loc);
   template <class charT> bool isalnum (charT c, const locale& loc);
   template <class charT> bool isgraph (charT c, const locale& loc);
   template <class charT> bool isblank (charT c, const locale& loc);
  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).<sup>238</sup>
   22.3.3.2 Conversions
                                                                                           [conversions]
   22.3.3.2.1 Character conversions
                                                                                [conversions.character]
   template <class charT> charT toupper(charT c, const locale& loc);
        Returns: use_facet<ctype<charT> >(loc).toupper(c).
   template <class charT> charT tolower(charT c, const locale& loc);
        Returns: use_facet<ctype<charT> >(loc).tolower(c).
   22.3.3.2.2 string conversions
                                                                                    [conversions.string]
  Class template wstring_convert performs conversions between a wide string and a byte string. It lets you
   specify a code conversion facet (like class template codecvt) to perform the conversions, without affecting
   any streams or locales. [Example: Say, for example, you have a code conversion facet called codecvt_utf8
   that you want to use to output to cout a UTF-8 multibyte sequence corresponding to a wide string, but
   you don't want to alter the locale for cout. You can write something like:
     wstring_convert<codecvt_utf8<wchar_t>> myconv;
     std::string mbstring = myconv.to_bytes(L"Hello\n");
     std::cout << mbstring;</pre>
   — 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;
         wstring_convert(Codecvt *pcvt = new Codecvt);
         wstring_convert(Codecvt *pcvt, state_type state);
```

const wide\_string& wide\_err = wide\_string());

wstring\_convert(const byte\_string& byte\_err,

wide\_string from\_bytes(char byte);
wide\_string from\_bytes(const char \*ptr);
wide\_string from\_bytes(const byte\_string& str);

~wstring\_convert();

§ 22.3.3.2.2 654

<sup>238)</sup> When used in a loop, it is faster to cache the ctype<> facet and use it directly, or use the vector form of ctype<>::is.

```
wide_string from_bytes(const char *first, const char *last);
         byte_string to_bytes(Elem wchar);
         byte_string to_bytes(const Elem *wptr);
         byte_string to_bytes(const wide_string& wstr);
         byte_string to_bytes(const Elem *first, const Elem *last);
         size t converted() const;
         state_type state() const;
       private:
                                          // exposition only
         byte_string byte_err_string;
                                          // exposition only
         wide_string wide_err_string;
                                          // exposition only
         Codecvt *cvtptr;
                                          // exposition only
         state_type cvtstate;
         size_t cvtcount;
                                          // exposition only
       };
     }
  The class template describes an object that controls conversions between wide string objects of class
   std::basic_string<Elem, char_traits<Elem>, Wide_alloc> and byte string objects of class std::
   basic_string<char, char_traits<char>, Byte_alloc>. The class template defines the types wide_-
   string and byte_string as synonyms for these two types. Conversion between a sequence of Elem values
   (stored in a wide_string object) and multibyte sequences (stored in a byte_string object) is performed by
   an object of class Codecvt<Elem, char, std::mbstate_t>, which meets the requirements of the standard
   code-conversion facet std::codecvt<Elem, char, std::mbstate_t>.
4 An object of this class template stores:

    byte_err_string — a byte string to display on errors

     — wide_err_string — a wide string to display on errors
     — cvtptr — a pointer to the allocated conversion object (which is freed when the wstring_convert
        object is destroyed)

    cvtstate — a conversion state object

     — cvtcount — a conversion count
   typedef std::basic_string<char> byte_string;
        The type shall be a synonym for std::basic_string<char>
   size_t converted() const;
        Returns: cvtcount.
   wide_string from_bytes(char byte);
```

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:

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);

5

6

7

§ 22.3.3.2.2 655

— If the cvtstate object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.

- The number of input elements successfully converted shall be stored in cvtcount.
- 9 Returns: If no conversion error occurs, the member function shall return the converted wide string. Otherwise, if the object was constructed with a wide-error string, the member function shall return the wide-error string. Otherwise, the member function throws an object of class std::range\_error.

```
typedef typename wide_string::traits_type::int_type int_type;

The type shall be a synonym for wide_string::traits_type::int_type.

state_type state() const;

returns cvtstate.

typedef typename Codecvt::state_type state_type;

The type shall be a synonym for Codecvt::state_type.

byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem *wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem *first, const Elem *last);
```

- Effects: The first member function shall convert the single-element sequence wchar to a byte string. The second member function shall convert the null-terminated sequence beginning at wptr to a byte string. The third member function shall convert the sequence stored in wstr to a byte string. The fourth member function shall convert the sequence defined by the range [first,last) to a byte string.
- In all cases:

12

- 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>.

§ 22.3.3.2.2 656

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();
```

17

Effects: The destructor shall delete cvtptr.

#### 22.3.3.2.3 Buffer conversions

[conversions.buffer]

- Class template wbuffer\_convert looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template wstring\_convert, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.
- Class template wbuffer\_convert synopsis

```
namespace std {
template < class Codecvt,
    class Elem = wchar_t,
    class Tr = std::char_traits<Elem> >
  class wbuffer_convert
    : public std::basic_streambuf<Elem, Tr> {
  public:
    typedef typename Codecvt::state_type state_type;
    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:
                                     // exposition only
    std::streambuf *bufptr;
    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 codeconversion facet std::codecvt<Elem, char, std::mbstate\_t>.
- 4 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

§ 22.3.3.2.3 657

```
state_type state() const;
5
         Returns: cvtstate.
    std::streambuf *rdbuf() const;
6
         Returns: bufptr.
    std::streambuf *rdbuf(std::streambuf *bytebuf);
7
         Effects: stores bytebuf in bufptr.
8
         Returns: The previous value of bufptr.
    typedef typename Codecvt::state_type state_type;
9
         The type shall be a synonym for Codecvt::state type.
    wbuffer_convert(std::streambuf *bytebuf = 0,
        Codecvt *pcvt = new Codecvt, state_type state = state_type());
10
         Effects: The constructor constructs a stream buffer object, initializes buffer to bytebuf, initializes
         cvtptr to pcvt, and initializes cvtstate to state.
    ~wbuffer_convert();
11
         Effects: The destructor shall delete cvtptr.
```

### 22.4 Standard locale categories

[locale.categories]

- Each of the standard categories includes a family of facets. Some of these implement formatting or parsing of a datum, for use by standard or users' iostream operators << and >>, as members put() and get(), respectively. Each such member function takes an ios\_base& argument whose members flags(), precision(), and width(), specify the format of the corresponding datum (27.5.3). Those functions which need to use other facets call its member getloc() to retrieve the locale imbued there. Formatting facets use the character argument fill to fill out the specified width where necessary.
- The put() members make no provision for error reporting. (Any failures of the OutputIterator argument must be extracted from the returned iterator.) The get() members take an ios\_base::iostate& argument whose value they ignore, but set to ios\_base::failbit in case of a parse error.
- 3 Within this clause it is unspecified whether one virtual function calls another virtual function.

## 22.4.1 The ctype category

[category.ctype]

```
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;</pre>
```

§ 22.4.1 658

```
static const mask lower = 1 << 4;
      static const mask alpha = 1 << 5;
      static const mask digit = 1 << 6;
      static const mask punct = 1 << 7;</pre>
      static const mask xdigit = 1 << 8;
      static const mask blank = 1 << 9;
      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
                                                                                        [locale.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);
                    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;
                   tolower(charT c) const;
       const charT* tolower(charT* low, const charT* high) const;
                    widen(char c) const;
      charT
       const char* widen(const char* low, const char* high, charT* to) const;
                   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,
```

§ 22.4.1.1 659

```
charT* dest) const;
                              do_narrow(charT, char dfault) const;
         virtual char
         virtual const charT* do_narrow(const charT* low, const charT* high,
                                        char dfault, char* dest) const;
      };
     }
Class ctype encapsulates the C library <cctype> features. istream members are required to use ctype<>
   for character classing during input parsing.
<sup>2</sup> The specializations required in Table 81 (22.3.1.1.1), namely ctype<char> and ctype<wchar_t>, implement
   character classing appropriate to the implementation's native character set.
   22.4.1.1.1 ctype members
                                                                               [locale.ctype.members]
   bool
                is(mask m, charT c) const;
   const charT* is(const charT* low, const charT* high,
                   mask* vec) const;
        Returns: do_is(m,c) or do_is(low,high,vec)
   const charT* scan_is(mask m,
                        const charT* low, const charT* high) const;
        Returns: do_scan_is(m,low,high)
   const charT* scan_not(mask m,
                         const charT* low, const charT* high) const;
        Returns: do_scan_not(m,low,high)
   charT
                toupper(charT) const;
   const charT* toupper(charT* low, const charT* high) const;
        Returns: do_toupper(c) or do_toupper(low,high)
                tolower(charT c) const;
   const charT* tolower(charT* low, const charT* high) const;
        Returns: do_tolower(c) or do_tolower(low,high)
               widen(char c) const;
   const char* widen(const char* low, const char* high, charT* to) const;
        Returns: do_widen(c) or do_widen(low,high,to)
                narrow(charT c, char dfault) const;
   const charT* narrow(const charT* low, const charT*, char dfault,
                       char* to) const;
```

2

3

5

6

660 § 22.4.1.1.1

Returns: do\_narrow(c,dfault) or do\_narrow(low,high,dfault,to)

#### 22.4.1.1.2 ctype virtual functions

[locale.ctype.virtuals]

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) != 0; i.e., true if the character has the characteristics specified. The second form returns high.

- 3 Effects: Locates a character in a buffer that conforms to a classification m.
- 4 Returns: The smallest pointer p in the range [low, high) such that is(m,\*p) would return true; otherwise, returns high.

- 5 Effects: Locates a character in a buffer that fails to conform to a classification m.
- Returns: The smallest pointer p, if any, in the range [low,high) such that is(m,\*p) would return false; otherwise, returns high.

```
charT do_toupper(charT c) const;
const charT* do_toupper(charT* low, const charT* high) const;
```

- Effects: Converts a character or characters to upper case. The second form replaces each character \*p in the range [low,high) for which a corresponding upper-case character exists, with that character.
- 8 Returns: The first form returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form returns high.

```
charT do_tolower(charT c) const;
const charT* do_tolower(charT* low, const charT* high) const;
```

- 9 Effects: Converts a character or characters to lower case. The second form replaces each character \*p in the range [low,high) and for which a corresponding lower-case character exists, with that character.
- Returns: The first form returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form returns high.

§ 22.4.1.1.2 661

Effects: Applies the simplest reasonable transformation from a char value or sequence of char values to the corresponding charT value or values.<sup>239</sup> The only characters for which unique transformations are required are those in the basic source character set (2.3).

For any named ctype category with a ctype<charT> facet ctc and valid ctype\_base::mask value M, (ctc.is(M, c) || !is(M, do\_widen(c)) ) is true.<sup>240</sup>

The second form transforms each character \*p in the range [low,high), placing the result in dest[p-low].

Returns: The first form returns the transformed value. The second form returns high.

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
```

12

13

For any named ctype category with a ctype  $\$  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

[locale.ctype.byname]

```
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
                                                                                [facet.ctype.special]
 namespace std {
    template <> class ctype<char>
      : public locale::facet, public ctype_base {
    public:
```

§ 22.4.1.3 662

<sup>239)</sup> The char argument of do\_widen is intended to accept values derived from character literals for conversion to the locale's encoding.

<sup>240)</sup> In other words, the transformed character is not a member of any character classification that c is not also a member of.

```
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;
                toupper(char c) const;
    char
    const char* toupper(char* low, const char* high) const;
                tolower(char c) const;
    const char* tolower(char* low, const char* high) const;
    char widen(char c) const;
    const char* widen(const char* low, const char* high, char* to) const;
    char narrow(char c, char dfault) const;
    const char* narrow(const char* low, const char* high, char dfault,
                       char* to) const;
    static locale::id id;
    static const size_t table_size = implementation-defined;
    const mask* table() const noexcept;
    static const mask* classic_table() noexcept;
  protected:
   ~ctype();
    virtual char
                        do_toupper(char c) const;
    virtual const char* do_toupper(char* low, const char* high) const;
    virtual char
                        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;
 };
}
```

A specialization ctype<char> is provided so that the member functions on type char can be implemented inline.<sup>241</sup> 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();

§ 22.4.1.3.1 663

<sup>241)</sup> 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>.

1 Effects: If the constructor's first argument was nonzero, and its second argument was true, does delete [] table().

```
22.4.1.3.2 ctype<char> members
```

explicit ctype(const mask\* tbl = 0, bool del = false,

# [facet.ctype.char.members]

In the following member descriptions, for unsigned char values v where v >= table\_size, table()[v] is assumed to have an implementation-specific value (possibly different for each such value v) without performing the array lookup.

```
size_t refs = 0);
2
        Requires: tbl either 0 or an array of at least table size elements.
3
        Effects: Passes its refs argument to its base class constructor.
   bool
               is(mask m, char c) const;
   const char* is(const char* low, const char* high,
                  mask* vec) const;
4
        Effects: The second form, for all *p in the range [low,high), assigns into vec[p-low] the value
        table()[(unsigned char)*p].
5
        Returns: The first form returns table()[(unsigned char)c] & m; the second form returns high.
   const char* scan_is(mask m,
                       const char* low, const char* high) const;
6
        Returns: The smallest p in the range [low,high) such that
          table()[(unsigned char) *p] & m
        is true.
   const char* scan_not(mask m,
                        const char* low, const char* high) const;
7
        Returns: The smallest p in the range [low,high) such that
          table()[(unsigned char) *p] & m
        is false.
               toupper(char c) const;
   char
   const char* toupper(char* low, const char* high) const;
8
        Returns: do_toupper(c) or do_toupper(low,high), respectively.
   char
               tolower(char c) const;
   const char* tolower(char* low, const char* high) const;
9
        Returns: do_tolower(c) or do_tolower(low,high), respectively.
```

§ 22.4.1.3.2

```
char widen(char c) const;
   const char* widen(const char* low, const char* high,
       char* to) const;
10
         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;
11
         Returns: do_narrow(c, dfault) or do_narrow(low, high, dfault, to), respectively.
   const mask* table() const noexcept;
12
         Returns: The first constructor argument, if it was non-zero, otherwise classic_table().
               ctype<char> static members
                                                                               [facet.ctype.char.statics]
   static const mask* classic_table() noexcept;
         Returns: A pointer to the initial element of an array of size table_size which represents the classifi-
         cations of characters in the "C" locale.
   22.4.1.3.4 ctype<char> virtual functions
                                                                              [facet.ctype.char.virtuals]
                  do_toupper(char) const;
      char
      const char* do_toupper(char* low, const char* high) const;
                  do_tolower(char) const;
      const char* do_tolower(char* low, const char* high) const;
                          do_widen(char c) const;
     virtual char
     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 tem-
   plate (22.4.1.1.1).
   22.4.1.4 Class template codecvt
                                                                                         [locale.codecvt]
     namespace std {
       class codecvt_base {
       public:
         enum result { ok, partial, error, noconv };
       template <class internT, class externT, class stateT>
       class codecvt : public locale::facet, public codecvt_base {
          typedef internT intern_type;
          typedef externT extern_type;
         typedef stateT state_type;
```

665

§ 22.4.1.4

```
explicit codecvt(size_t refs = 0);
  result out(stateT& state,
             const internT* from, const internT* from_end, const internT*& from_next,
                                 externT* to_end, externT*& to_next) const;
             externT* to,
  result unshift(stateT& state,
                 externT* to,
                                       externT* to_end, externT*& to_next) const;
  result in(stateT& state,
            const externT* from, const externT* from_end, const externT*& from_next,
            internT* to,
                                 internT* to_end, internT*& to_next) const;
  int encoding() const noexcept;
  bool always_noconv() const noexcept;
  int length(stateT&, const externT* from, const externT* end,
             size_t max) const;
  int max_length() const noexcept;
  static locale::id id;
protected:
  ~codecvt();
  virtual result do_out(stateT& state,
                        const internT* from, const internT* from_end, const internT*& from_next,
                        externT* to,
                                             externT* to_end, externT*& to_next) const;
  virtual result do_in(stateT& state,
                       const externT* from, const externT* from_end, const externT*& from_next,
                                           internT* to_end, internT*& to_next) const;
                       internT* to,
  virtual result do_unshift(stateT& state,
                            externT* to,
                                                 externT* to_end, externT*& to_next) const;
  virtual int do_encoding() const noexcept;
  virtual bool do_always_noconv() const noexcept;
  virtual int do_length(stateT&, const externT* from,
                        const externT* end, size_t max) const;
  virtual int do_max_length() const noexcept;
};
```

- 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 81 (22.3.1.1.1) convert the implementation-defined native character set. codecvt<char, char, mbstate\_t> implements a degenerate conversion; it does not convert at all. The specialization codecvt<char16\_t, char, mbstate\_t> converts between the UTF-16 and UTF-8 encoding 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 implementer. 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

[locale.codecvt.members]

```
result out(stateT& state,
  const internT* from, const internT* from_end, const internT*& from_next,
```

§ 22.4.1.4.1 666

```
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;
2
        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;
3
        Returns: do_in(state, from, from_end, from_next, to, to_end, to_next)
   int encoding() const noexcept;
        Returns: do_encoding()
   bool always_noconv() const noexcept;
        Returns: do_always_noconv()
   int length(stateT& state, const externT* from, const externT* from_end,
              size_t max) const;
        Returns: do_length(state, from,from_end,max)
   int max_length() const noexcept;
        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 be-
        ginning 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).
        A codecvt facet that is used by basic_filebuf (27.9) shall have the property that if
```

§ 22.4.1.4.2

```
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.  $^{242}$  [Note: As a result of operations on state, it can return ok or partial and set from next == from and to next != to. — end note]

- Remarks: Its operations on state are unspecified. [Note: This argument can be used, for example, to maintain shift state, to specify conversion options (such as count only), or to identify a cache of seek offsets. —end note]
- 5 Returns: An enumeration value, as summarized in Table 83.

Value	Meaning
ok	completed the conversion
partial	not all source characters converted
error	encountered a character in [from,from_end) that
	it could not convert
noconv	internT and externT are the same type, and in-
	put sequence is identical to converted sequence

Table 83 — do\_in/do\_out result values

A return value of partial, if (from\_next==from\_end), indicates that either the destination sequence has not absorbed all the available destination elements, or that additional source elements are needed before another destination element can be produced.

```
result do_unshift(stateT& state,
  externT* to, externT* to_end, externT*& to_next) const;
```

- Requires: (to <= to\_end) well defined and true; state initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.
- 7 Effects: Places characters starting at to that should be appended to terminate a sequence when the current stateT is given by state.<sup>243</sup> 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 84.

int do\_encoding() const noexcept;

§ 22.4.1.4.2

<sup>242)</sup> 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.

243) Typically these will be characters to return the state to stateT()

Table 84 — do unshift n	result values
-------------------------	---------------

Value	Meaning
ok	completed the sequence
partial	space for more than to_end-to destination elements was needed to terminate a sequence given the value of
	state
error	an unspecified error has occurred
noconv	no termination is needed for this state_type

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.<sup>244</sup>

bool do\_always\_noconv() const noexcept;

Returns: true if do\_in() and do\_out() return noconv for all valid argument values. codecvt<char, char, mbstate\_t> returns true.

- Requires: (from<=from\_end) well-defined and true; state initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.
- Effects: The effect on the state argument is "as if" it called do\_in(state, from, from\_end, from, to, to+max, to) for to pointing to a buffer of at least max elements.
- Returns: (from\_next-from) where from\_next is the largest value in the range [from,from\_end] such that the sequence of values in the range [from,from\_next) represents max or fewer valid complete characters of type internT. The specialization codecvt<char, char, mbstate\_t>, returns the lesser of max and (from\_end-from).

int do\_max\_length() const noexcept;

Returns: The maximum value that do\_length(state, from, from\_end, 1) can return for any valid range [from, from\_end) and stateT value state. The specialization codecvt<char, char, mbstate\_t>::do\_max\_length() returns 1.

#### 22.4.1.5 Class template codecvt byname

[locale.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:
```

§ 22.4.1.5

<sup>244)</sup> 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.

```
~codecvt_byname();
};
```

# 22.4.2 The numeric category

[category.numeric]

The classes num\_get<> and num\_put<> handle numeric formatting and parsing. Virtual functions are provided for several numeric types. Implementations may (but are not required to) delegate extraction of smaller types to extractors for larger types.<sup>245</sup>

- All specifications of member functions for num\_put and num\_get in the subclauses of 22.4.2 only apply to the specializations required in Tables 81 and 82 (22.3.1.1.1), namely num\_get<char>, num\_get<wchar\_t>, num\_get<C, InputIterator>, num\_put<char>, num\_put<wchar\_t>, and num\_put<C,OutputIterator>. These specializations refer to the ios\_base& argument for formatting specifications (22.4), and to its imbued locale for the numpunct<> facet to identify all numeric punctuation preferences, and also for the ctype<> facet to perform character classification.
- Extractor and inserter members of the standard iostreams use num\_get<> and num\_put<> member functions for formatting and parsing numeric values (27.7.2.2.1, 27.7.3.6.1).

# 22.4.2.1 Class template num\_get

[locale.num.get]

```
namespace std {
  template <class charT, class InputIterator = istreambuf iterator<charT> >
  class num_get : public locale::facet {
 public:
    typedef charT
                             char_type;
    typedef InputIterator
                             iter_type;
    explicit num_get(size_t refs = 0);
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, bool& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, long long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned short& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned int& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned long long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, float& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, double& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, long double& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, void*& v) const;
    static locale::id id;
```

§ 22.4.2.1 670

<sup>245)</sup> Parsing "-1" correctly into, e.g., an unsigned short requires that the corresponding member get() at least extract the sign before delegating.

```
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
                                                                          [facet.num.get.members]
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, bool& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned short& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned int& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, float& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, void*& val) const;
      Returns: do_get(in, end, str, err, val).
22.4.2.1.2 num get virtual functions
                                                                            [facet.num.get.virtuals]
```

§ 22.4.2.1.2 671

```
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
 ios_base::iostate& err, unsigned short& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
 ios_base::iostate& err, unsigned int& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
 ios_base::iostate& err, unsigned long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
 ios_base::iostate& err, unsigned long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, float& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, void*& val) const;
     Effects: Reads characters from in, interpreting them according to str.flags(), use_facet<ctype<
     charT> >(loc), and use facet< numpunct<charT> >(loc), where loc is str.getloc().
```

- The details of this operation occur in three stages
  - Stage 1: Determine a conversion specifier
  - Stage 2: Extract characters from in and determine a corresponding char value for the format expected by the conversion specification determined in stage 1.
  - Stage 3: Store results
- The details of the stages are presented below.

Stage 1: The function initializes local variables via

```
fmtflags flags = str .flags();
fmtflags basefield = (flags & ios_base::basefield);
fmtflags uppercase = (flags & ios_base::uppercase);
fmtflags boolalpha = (flags & ios_base::boolalpha);
```

For conversion to an integral type, the function determines the integral conversion specifier as indicated in Table 85. The table is ordered. That is, the first line whose condition is true applies.

Statestdio equivalentbasefield == oct%obasefield == hex%Xbasefield == 0%isigned integral type%dunsigned integral type%u

Table 85 — Integer conversions

For conversions to a floating type the specifier is %g.

For conversions to void\* the specifier is %p.

A length modifier is added to the conversion specification, if needed, as indicated in Table 86.

§ 22.4.2.1.2 672

Type	Length modifier
short	h
unsigned short	h
long	1
unsigned long	1
long long	11
unsigned long long	11
double	1
long double	L

Table 86 — Length modifier

Stage 2: If in==end then stage 2 terminates. Otherwise a charT is taken from in and local variables are initialized as if by

```
char_type ct = *in ;
  char c = src[find(atoms, atoms + sizeof(src) - 1, ct) - atoms];
  if (ct == use_facet<numpunct<charT> >(loc).decimal_point())
  c = '.';
  bool discard =
     ct == use_facet<numpunct<charT> >(loc).thousands_sep()
     && use_facet<numpunct<charT> >(loc).grouping().length() != 0;

where the values src and atoms are defined as if by:
  static const char src[] = "0123456789abcdefxABCDEFX+-";
  char_type atoms[sizeof(src)];
  use_facet<ctype<charT> >(loc).widen(src, src + sizeof(src), atoms);
```

for this value of loc.

If discard is true, then if '.' has not yet been accumulated, then the position of the character is remembered, but the character is otherwise ignored. Otherwise, if '.' has already been accumulated, the character is discarded and Stage 2 terminates. If it is not discarded, then a check is made to determine if c is allowed as the next character of an input field of the conversion specifier returned by Stage 1. If so, it is accumulated.

If the character is either discarded or accumulated then in is advanced by ++in and processing returns to the beginning of stage 2.

**Stage 3:** The sequence of chars accumulated in stage 2 (the field) is converted to a numeric value by the rules of one of the functions declared in the header <cstdlib>:

- For a signed integer value, the function strtoll.
- For an unsigned integer value, the function strtoull.
- For a floating-point value, the function strtold.

The numeric value to be stored can be one of:

- zero, if the conversion function fails to convert the entire field. ios\_base::failbit is assigned to err.
- the most positive representable value, if the field represents a value too large positive to be represented in val. ios\_base::failbit is assigned to err.
- the most negative representable value or zero for an unsigned integer type, if the field represents a value too large negative to be represented in val. ios\_base::failbit is assigned to err.

§ 22.4.2.1.2 673

— the converted value, otherwise.

The resultant numeric value is stored in val.

Digit grouping is checked. That is, the positions of discarded separators is examined for consistency with use\_facet<numpunct<charT> >(loc).grouping(). If they are not consistent then ios\_base::failbit is assigned to err.

In any case, if stage 2 processing was terminated by the test for in==end then err |=ios\_base::eofbit is performed.

- Effects: If (str.flags()&ios\_base::boolalpha)==0 then input proceeds as it would for a long except that if a value is being stored into val, the value is determined according to the following: If the value to be stored is 0 then false is stored. If the value is 1 then true is stored. Otherwise true is stored and ios\_base::failbit is assigned to err.
- Otherwise target sequences are determined "as if" by calling the members falsename() and truename() of the facet obtained by use\_facet<numpunct<charT> >(str.getloc()). Successive characters in the range [in,end) (see 23.2.3) are obtained and matched against corresponding positions in the target sequences only as necessary to identify a unique match. The input iterator in is compared to end only when necessary to obtain a character. If 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]
- 9 Returns: in.

#### 22.4.2.2 Class template num\_put

[locale.nm.put]

```
namespace std {
 template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
 class num_put : public locale::facet {
 public:
    typedef charT
                             char_type;
    typedef OutputIterator
                            iter_type;
   explicit num_put(size_t refs = 0);
   iter_type put(iter_type s, ios_base& f, char_type fill, bool v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  unsigned long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  unsigned long long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
```

§ 22.4.2.2 674

```
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;
      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
                                                                          [facet.num.put.members]
iter_type put(iter_type out, ios_base& str, char_type fill,
  bool val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  unsigned long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  unsigned long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  long double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  const void* val) const;
     Returns: do_put(out, str, fill, val).
                                                                            [facet.num.put.virtuals]
22.4.2.2.2 num_put virtual functions
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
§ 22.4.2.2.2
                                                                                                 675
```

```
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
   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.
- 3 Detailed descriptions of each stage follow.
- 4 Returns: out.

5

**Stage 1:** The first action of stage 1 is to determine a conversion specifier. The tables that describe this determination use the following local variables

```
fmtflags flags = str.flags() ;
fmtflags basefield = (flags & (ios_base::basefield));
fmtflags uppercase = (flags & (ios_base::uppercase));
fmtflags floatfield = (flags & (ios_base::floatfield));
fmtflags showpos = (flags & (ios_base::showpos));
fmtflags showbase = (flags & (ios_base::showbase));
```

All tables used in describing stage 1 are ordered. That is, the first line whose condition is true applies. A line without a condition is the default behavior when none of the earlier lines apply.

For conversion from an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 87.

For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 88.

For conversions from an integral or floating-point type a length modifier is added to the conversion specifier as indicated in Table 89.

The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 90.

§ 22.4.2.2.2

Table 87 — Integer conversions

State	stdio equivalent
basefield == ios_base::oct	<b>%</b> o
(basefield == ios_base::hex) && !uppercase	%x
(basefield == ios_base::hex)	%X
for a signed integral type	%d
for an unsigned integral type	%u

Table 88 — Floating-point conversions

State	stdio equivalent
floatfield == ios_base::fixed	%f
floatfield == ios_base::scientific && !uppercase	%e
floatfield == ios_base::scientific	%E
floatfield == (ios_base::fixed   ios_base::scientific) && !uppercase	%a
floatfield == (ios_base::fixed   ios_base::scientific)	%A
!uppercase	%g
otherwise	%G

Table 89 — Length modifier

Type	Length modifier
long	1
long long	11
unsigned long	1
unsigned long long	11
long double	L
otherwise	none

Table 90 — Numeric conversions

Type(s)	State	stdio equivalent
an integral type	flags & showpos	+
	flags & showbase	#
a floating-point type	flags & showpos	+
	flags & showpoint	#

§ 22.4.2.2.2

For conversion from a floating-point type, if floatfield != (ios\_base::fixed | ios\_base::scientific), str.precision() is specified as precision in the conversion specification. Otherwise, no precision is specified.

For conversion from void\* the specifier is %p.

The representations at the end of stage 1 consists of the char's that would be printed by a call of printf(s, val) where s is the conversion specifier determined above.

Stage 2: Any character c other than a decimal point(.) is converted to a charT via use\_facet<ctype<charT> >(loc).widen( c )

A local variable punct is initialized via

```
const numpunct<charT>& punct = use_facet< numpunct<charT> >(str.getloc());
```

For arithmetic types, punct.thousands\_sep() characters are inserted into the sequence as determined by the value returned by punct.do\_grouping() using the method described in 22.4.3.1.2 Decimal point characters(.) are replaced by punct.decimal\_point()

Stage 3: A local variable is initialized as

```
fmtflags adjustfield= (flags & (ios_base::adjustfield));
```

The location of any padding $^{246}$  is determined according to Table 91.

Table 91 —	- Fill	padding

State	Location
adjustfield == ios_base::left	pad after
adjustfield == ios_base::right	pad before
adjustfield == internal and a sign occurs in	pad after the sign
the representation	
adjustfield == internal and representation	pad after x or X
after stage 1 began with 0x or 0X	
otherwise	pad before

If str.width() is nonzero and the number of charT's in the sequence after stage 2 is less than str.width(), then enough fill characters are added to the sequence at the position indicated for padding to bring the length of the sequence to str.width().

str.width(0) is called.

Stage 4: The sequence of chart's at the end of stage 3 are output via

```
*out++ = c
```

and then inserts each character c of s into out via \*out++ = c and returns out.

§ 22.4.2.2.2 678

### 22.4.3 The numeric punctuation facet

# [facet.numpunct] [locale.numpunct]

#### 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;
                 grouping()
                                    const;
    string
    string_type truename()
                                    const;
    string_type falsename()
                                    const;
    static locale::id id;
 protected:
                                 // virtual
   ~numpunct();
                         do_decimal_point() const;
    virtual char_type
    virtual char_type
                         do_thousands_sep() const;
    virtual string
                         do_grouping()
                                             const:
                                                          // for bool
    virtual string_type do_truename()
                                             const:
    virtual string_type do_falsename()
                                                          // for bool
                                             const;
  };
}
```

- numpunct<> specifies numeric punctuation. The specializations required in Table 81 (22.3.1.1.1), namely numpunct<wchar\_t> and numpunct<char>, provide classic "C" numeric formats, i.e., they contain information equivalent to that contained in the "C" locale or their wide character counterparts as if obtained by a call to widen.
- The syntax for number formats is as follows, where digit represents the radix set specified by the fmtflags argument value, and thousands-sep and decimal-point are the results of corresponding numpunct<charT> members. Integer values have the format:

where the number of digits between thousands-seps is as specified by do\_grouping(). For parsing, if the digits portion contains no thousands-separators, no grouping constraint is applied.

## 22.4.3.1.1 numpunct members

[facet.numpunct.members]

```
char_type decimal_point() const;

Returns: do_decimal_point()
```

§ 22.4.3.1.1 679

```
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.
                                                                                [facet.numpunct.virtuals]
   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'.'.
   char_type do_thousands_sep() const;
2
        Returns: A character for use as the digit group separator. The required specializations return ',' or
        L','.
   string do_grouping() const;
3
        Returns: A basic string<char> vec used as a vector of integer values, in which each element vec[i]
        represents the number of digits^{247} in the group at position i, starting with position 0 as the rightmost
        group. If vec.size() <= i, the number is the same as group (i-1); if (i<0 || vec[i]<=0 ||
        vec[i] == CHAR_MAX), the size of the digit group is unlimited.
4
        The required specializations return the empty string, indicating no grouping.
                               const;
   string_type do_truename()
   string_type do_falsename() const;
5
        Returns: A string representing the name of the boolean value true or false, respectively.
6
        In the base class implementation these names are "true" and "false", or L"true" and L"false".
   247) Thus, the string "\003" specifies groups of 3 digits each, and "3" probably indicates groups of 51 (!) digits each, because
```

§ 22.4.3.1.2

51 is the ASCII value of "3".

```
Class template numpunct_byname
                                                                            [locale.numpunct.byname]
     namespace std {
       template <class charT>
       class numpunct_byname : public numpunct<charT> {
       // this class is specialized for char and wchar_t.
       public:
         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
                                                                                     [category.collate]
   22.4.4.1
             Class template collate
                                                                                         [locale.collate]
     namespace std {
       template <class charT>
       class collate : public locale::facet {
       public:
         typedef charT
                                      char_type;
         typedef basic_string<charT> string_type;
         explicit collate(size_t refs = 0);
         int compare(const charT* low1, const charT* high1,
                     const charT* low2, const charT* high2) const;
         string_type transform(const charT* low, const charT* high) const;
         long hash(const charT* low, const charT* high) const;
         static locale::id id;
       protected:
         ~collate();
         virtual int do_compare(const charT* low1, const charT* high1,
                                const charT* low2, const charT* high2) const;
         virtual string_type do_transform(const charT* low, const charT* high) const;
         virtual long do_hash (const charT* low, const charT* high) const;
       };
     }
  The class collate<charT> provides features for use in the collation (comparison) and hashing of strings.
   A locale member function template, operator(), uses the collate facet to allow a locale to act directly
   as the predicate argument for standard algorithms (Clause 25) and containers operating on strings. The
   specializations required in Table 81 (22.3.1.1.1), namely collate<char> and collate<wchar_t>, apply
   lexicographic ordering (25.4.8).
<sup>2</sup> Each function compares a string of characters *p in the range [low,high).
   22.4.4.1.1 collate members
                                                                               [locale.collate.members]
   int compare(const charT* low1, const charT* high1,
               const charT* low2, const charT* high2) const;
        Returns: do_compare(low1, high1, low2, high2)
   § 22.4.4.1.1
                                                                                                     681
```

```
string_type transform(const charT* low, const charT* high) const;
2
        Returns: do_transform(low, high)
   long hash(const charT* low, const charT* high) const;
3
        Returns: do_hash(low, high)
   22.4.4.1.2 collate virtual functions
                                                                                 [locale.collate.virtuals]
   int do_compare(const charT* low1, const charT* high1,
                  const charT* low2, const charT* high2) const;
1
        Returns: 1 if the first string is greater than the second, -1 if less, zero otherwise. The specializa-
        tions required in Table 81 (22.3.1.1.1), namely collate<char> and collate<wchar_t>, implement a
        lexicographical comparison (25.4.8).
   string_type do_transform(const charT* low, const charT* high) const;
2
        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.<sup>248</sup>
   long do_hash(const charT* low, const charT* high) const;
3
        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/numeric\_limits < unsigned long > :: max()). - end note]
   22.4.4.2
             Class template collate_byname
                                                                                [locale.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

[category.time]

Templates time\_get<charT,InputIterator> and time\_put<charT,OutputIterator> provide date and time formatting and parsing. All specifications of member functions for time\_put and time\_get in the subclauses of 22.4.5 only apply to the specializations required in Tables 81 and 82 (22.3.1.1.1). Their members use their ios\_base&, ios\_base::iostate&, and fill arguments as described in (22.4), and the ctype<> facet, to determine formatting details.

#### 22.4.5.1 Class template time\_get

[locale.time.get]

§ 22.4.5.1 682

<sup>248)</sup> This function is useful when one string is being compared to many other strings.

```
namespace std {
  class time_base {
  public:
    enum dateorder { no_order, dmy, mdy, ymd, ydm };
  };
  template <class charT, class InputIterator = istreambuf_iterator<charT> >
  class time_get : public locale::facet, public time_base {
 public:
    typedef charT
                             char_type;
    typedef InputIterator
                             iter_type;
    explicit time_get(size_t refs = 0);
    dateorder date_order() const { return do_date_order(); }
    iter_type get_time(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get_date(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get_weekday(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get_monthname(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get_year(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm *t, char format, char modifier = 0) const;
    iter_type get(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm *t, 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;
 };
}
```

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

§ 22.4.5.1 683

```
or unspecified values are assigned.<sup>249</sup>
<sup>2</sup> If the end iterator is reached during parsing by any of the get() member functions, the member sets ios_-
   base::eofbit in err.
   22.4.5.1.1 time_get members
                                                                             [locale.time.get.members]
   dateorder date_order() const;
1
        Returns: do_date_order()
   iter_type get_time(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
2
        Returns: do_get_time(s, end, str, err, t)
   iter_type get_date(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
3
        Returns: do_get_date(s, end, str, err, t)
   iter_type get_weekday(iter_type s, iter_type end, ios_base& str,
                         ios_base::iostate& err, tm* t) const;
   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)
   iter_type get(iter_type s, iter_type end, ios_base& f,
       ios_base::iostate& err, tm *t, char format, char modifier = 0) const;
6
        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.
8
        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.
```

§ 22.4.5.1.1 684

<sup>249)</sup> 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.

The next element of fmt is equal to '%', optionally followed by a modifier character, followed by a conversion specifier character, format, together forming a conversion specification valid for the ISO/IEC 9945 function strptime. If the number of elements in the range [fmt,fmtend) is not sufficient to unambiguously determine whether the conversion specification is complete and valid, the function evaluates err = ios\_base::failbit. Otherwise, the function evaluates s = do\_get(s, end, f, err, t, format, modifier), where the value of modifier is '\0' when the optional modifier is absent from the conversion specification. If err == ios\_base::goodbit holds after the evaluation of the expression, the function increments fmt to point just past the end of the conversion specification and continues looping.

- 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.
- 10 Returns: s

1

#### 22.4.5.1.2 time\_get virtual functions

[locale.time.get.virtuals]

dateorder do\_date\_order() const;

Returns: An enumeration value indicating the preferred order of components for those date formats that are composed of day, month, and year.<sup>250</sup> Returns no\_order if the date format specified by 'x' contains other variable components (e.g., Julian day, week number, week day).

- 2 Effects: Reads characters starting at s until it has extracted those struct tm members, and remaining format characters, used by time\_put<>::put to produce the format specified by "%H:%M:%S", or until it encounters an error or end of sequence.
- 3 Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid time.

- 4 Effects: Reads characters starting at s until it has extracted those struct tm members and remaining format characters used by time\_put<>::put to produce one of the following formats, or until it encounters an error. The format depends on the value returned by date\_order() as shown in Table 92.
- <sup>5</sup> An implementation may also accept additional implementation-defined formats.
- 6 Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid date.

§ 22.4.5.1.2

<sup>250)</sup> This function is intended as a convenience only, for common formats, and may return no\_order in valid locales.

Table 92 — d	do_get_	date	effects
--------------	---------	------	---------

date_order()	Format
no_order	"%m%d%y"
dmy	"%d%m%y"
mdy	"%m%d%y"
ymd	"%y%m%d"
ydm	"%y%d%m"

- 7 Effects: Reads characters starting at s until it has extracted the (perhaps abbreviated) name of a weekday or month. If it finds an abbreviation that is followed by characters that could match a full name, it continues reading until it matches the full name or fails. It sets the appropriate struct tm member accordingly.
- 8 Returns: An iterator pointing immediately beyond the last character recognized as part of a valid name.

- Effects: Reads characters starting at s until it has extracted an unambiguous year identifier. It is implementation-defined whether two-digit year numbers are accepted, and (if so) what century they are assumed to lie in. Sets the t->tm\_year member accordingly.
- Returns: An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

```
iter_type do_get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm *t, char format, char modifier) const;
```

- 11 Requires: t shall be dereferenceable.
- Effects: The function starts by evaluating err = ios\_base::goodbit. It then reads characters starting at s until it encounters an error, or until it has extracted and assigned those struct tm members, and any remaining format characters, corresponding to a conversion directive appropriate for the ISO/IEC 9945 function strptime, formed by concatenating '%', the modifier character, when non-NUL, and the format character. When the concatenation fails to yield a complete valid directive the function leaves the object pointed to by t unchanged and evaluates err |= ios\_base::failbit. When s == end evaluates to true after reading a character the function evaluates err |= ios\_-base::eofbit.
- For complex conversion directives such as %c, %x, or %X, or directives that involve the optional modifiers E or 0, when the function is unable to unambiguously determine some or all struct tm members from the input sequence [s,end), it evaluates err |= ios\_base::eofbit. In such cases the values of those struct tm members are unspecified and may be outside their valid range.

§ 22.4.5.1.2

Remark: It is unspecified whether multiple calls to do\_get() with the address of the same struct tm object will update the current contents of the object or simply overwrite its members. Portable programs must zero out the object before invoking the function.

Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid input sequence for the given format and modifier.

```
22.4.5.2 Class template time_get_byname
```

15

[locale.time.get.byname]

```
namespace std {
    template <class charT, class InputIterator = istreambuf_iterator<charT> >
    class time_get_byname : public time_get<charT, InputIterator> {
      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();
    };
  }
          Class template time_put
                                                                                    [locale.time.put]
22.4.5.3
 namespace std {
    template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
    class time_put : public locale::facet {
      typedef charT
                               char_type;
      typedef OutputIterator
                               iter_type;
      explicit time_put(size_t refs = 0);
      // the following is implemented in terms of other member functions.
      iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb,
                    const charT* pattern, const charT* pat_end) const;
      iter_type put(iter_type s, ios_base& f, char_type fill,
                    const tm* tmb, char format, char modifier = 0) const;
      static locale::id id;
   protected:
      ~time_put();
      virtual 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
                                                                         [locale.time.put.members]
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              char format, char modifier = 0) const;
     Effects: The first form steps through the sequence from pattern to pat_end, identifying characters
     that are part of a format sequence. Each character that is not part of a format sequence is written
```

§ 22.4.5.3.1 687

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<sup>251</sup> and a format specifier character spec as defined for the function strftime. If no modifier character is present, mod is zero. For each valid format sequence identified, calls do\_put(s, str, fill, t, spec, mod).

- The second form calls do\_put(s, str, fill, t, format, modifier).
- [Note: The fill argument may be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. end note]
- 4 Returns: An iterator pointing immediately after the last character produced.

#### 22.4.5.3.2 time\_put virtual functions

1

2

[locale.time.put.virtuals]

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()<sup>252</sup>, except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.<sup>253</sup>

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

[locale.time.put.byname]

### 22.4.6 The monetary category

[category.monetary]

These templates handle monetary formats. A template parameter indicates whether local or international monetary formats are to be used.

§ 22.4.6 688

<sup>251)</sup> Although the C programming language defines no modifiers, most vendors do.

<sup>252)</sup> Interpretation of the modifier argument is implementation-defined, but should follow POSIX conventions.

<sup>253)</sup> Implementations are encouraged to refer to other standards such as POSIX for these definitions.

All specifications of member functions for money\_put and money\_get in the subclauses of 22.4.6 only apply to the specializations required in Tables 81 and 82 (22.3.1.1.1). Their members use their ios\_base&, ios\_-base::iostate&, and fill arguments as described in (22.4), and the moneypunct<> and ctype<> facets, to determine formatting details.

```
22.4.6.1 Class template money_get
```

1

[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;
    };
                                                                      [locale.money.get.members]
22.4.6.1.1 money_get members
iter_type get(iter_type s, iter_type end, bool intl,
              ios_base& f, ios_base::iostate& err,
             long double& quant) const;
iter_type get(s, iter_type end, bool intl, ios_base&f,
              ios_base::iostate& err, string_type& quant) const;
     Returns: do_get(s, end, intl, f, err, quant)
22.4.6.1.2 money get virtual functions
                                                                        [locale.money.get.virtuals]
iter_type do_get(iter_type s, iter_type end, bool intl,
                 ios_base& str, ios_base::iostate& err,
                 long double& units) const;
iter_type do_get(iter_type s, iter_type end, bool intl,
                 ios_base& str, ios_base::iostate& err,
                 string_type& digits) const;
     Effects: Reads characters from s to parse and construct a monetary value according to the for-
     mat specified by a moneypunct<charT, Intl> facet reference mp and the character mapping spec-
```

§ 22.4.6.1.2 689

ified 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 units is produced as if by<sup>254</sup>

```
for (int i = 0; i < n; ++i)
  buf2[i] = src[find(atoms, atoms+sizeof(src), buf1[i]) - atoms];
buf2[n] = 0;
sscanf(buf2, "%Lf", &units);</pre>
```

where n is the number of characters placed in buf1, buf2 is a character buffer, and the values src and atoms are defined as if by

```
static const char src[] = "0123456789-";
charT atoms[sizeof(src)];
ct.widen(src, src + sizeof(src) - 1, atoms);
```

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

#### 22.4.6.2 Class template money\_put

[locale.money.put]

```
namespace std {
  template <class charT,
    class OutputIterator = ostreambuf_iterator<charT> >
  class money_put : public locale::facet {
```

§ 22.4.6.2

<sup>254)</sup> The semantics here are different from ct.narrow.

char\_type;

iter type;

public:

1

2

§ 22.4.6.2.2

typedef charT

typedef OutputIterator

typedef basic\_string<charT> string\_type;

```
explicit money_put(size_t refs = 0);
      iter_type put(iter_type s, bool intl, ios_base& f,
                    char_type fill, long double units) const;
      iter_type put(iter_type s, bool intl, ios_base& f,
                    char_type fill, const string_type& digits) const;
      static locale::id id;
    protected:
      ~money_put();
      virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                                long double units) const;
      virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                                const string_type& digits) const;
22.4.6.2.1 money_put members
                                                                        [locale.money.put.members]
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill,
              long double quant) const;
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill,
              const string_type& quant) const;
     Returns: do_put(s, intl, f, loc, quant)
22.4.6.2.2 money put virtual functions
                                                                          [locale.money.put.virtuals]
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, long double units) const;
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, const string_type& digits) const;
     Effects: Writes characters to s according to the format specified by a moneypunct<charT,Intl> facet
     reference mp and the character mapping specified by a ctype<charT> facet reference ct obtained from
     the locale returned by str.getloc(), and str.flags(). The argument units is transformed into a
     sequence of wide characters as if by
       ct.widen(buf1, buf1 + sprintf(buf1, "%.OLf", 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 subse-
     quent 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)

691

is true, the fill characters are placed where none or space appears in the formatting pattern; otherwise if (af == str.left) is true, they are placed after the other characters; otherwise, they are placed before the other characters. [Note: It is possible, with some combinations of format patterns and flag values, to produce output that cannot be parsed using num\_get<>::get. — end note]

Returns: An iterator pointing immediately after the last character produced.

#### 22.4.6.3 Class template moneypunct

3

[locale.moneypunct]

```
namespace std {
  class money_base {
 public:
    enum part { none, space, symbol, sign, value };
    struct pattern { char field[4]; };
  };
  template <class charT, bool International = false>
  class moneypunct : public locale::facet, public money_base {
  public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;
    explicit moneypunct(size_t refs = 0);
    charT
                 decimal_point() const;
    charT
                 thousands_sep() const;
    string
                 grouping()
                                 const;
    string_type curr_symbol()
                                 const;
    string_type positive_sign() const;
    string_type negative_sign() const;
                 frac digits()
    int
                                 const:
                 pos_format()
    pattern
                                  const:
   pattern
                 neg_format()
                                 const;
    static locale::id id;
    static const bool intl = International;
  protected:
    ~moneypunct();
    virtual charT
                         do_decimal_point() const;
    virtual charT
                         do_thousands_sep() const;
    virtual string
                         do_grouping()
                                             const;
    virtual string_type do_curr_symbol()
                                             const;
    virtual string_type do_positive_sign() const;
    virtual string_type do_negative_sign() const;
    virtual int
                         do_frac_digits()
                                             const;
    virtual pattern
                         do_pos_format()
                                             const;
    virtual pattern
                         do_neg_format()
                                             const;
  };
}
```

The moneypunct<> facet defines monetary formatting parameters used by money\_get<> and money\_put<>.

A monetary format is a sequence of four components, specified by a pattern value p, such that the part value static\_cast<part>(p.field[i]) determines the ith component of the format<sup>255</sup> In the field member of

§ 22.4.6.3

<sup>255)</sup> An array of char, rather than an array of part, is specified for pattern::field purely for efficiency.

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.
- 3 The format of the numeric monetary value is a decimal number:

```
value ::= units [ decimal-point [ digits ]] |
  decimal-point digits

if frac_digits() returns a positive value, or
value ::= units
```

otherwise. The symbol decimal-point indicates the character returned by decimal\_point(). The other symbols are defined as follows:

```
units ::= digits [ thousands-sep units ]
digits ::= adigit [ digits ]
```

In the syntax specification, the symbol adigit is any of the values ct.widen(c) for c in the range '0' through '9', inclusive, and ct is a reference of type const ctype<charT>& obtained as described in the definitions of money\_get<> and money\_put<>. The symbol thousands-sep is the character returned by thousands\_sep(). The space character used is the value ct.widen(' '). White space characters are those characters c for which ci.is(space,c) returns true. The number of digits required after the decimal point (if any) is exactly the value returned by frac\_digits().

The placement of thousands-separator characters (if any) is determined by the value returned by grouping(), defined identically as the member numpunct<>::do\_grouping().

#### 22.4.6.3.1 moneypunct members

[locale.moneypunct.members]

```
decimal_point() const;
charT
charT
             thousands_sep() const;
string
              grouping()
                              const;
string_type
             curr_symbol()
                              const;
             positive_sign() const;
string_type
string_type negative_sign() const;
             frac_digits()
int
                              const:
             pos_format()
pattern
                              const;
pattern
             neg_format()
                              const;
```

Each of these functions F returns the result of calling the corresponding virtual member function  ${\tt do\_F()}$ .

#### 22.4.6.3.2 moneypunct virtual functions

[locale.moneypunct.virtuals]

```
charT do_decimal_point() const;
```

1

Returns: The radix separator to use in case do\_frac\_digits() is greater than zero.<sup>256</sup>

charT do\_thousands\_sep() const;

2 Returns: The digit group separator to use in case do\_grouping() specifies a digit grouping pattern.<sup>257</sup>

```
256) In common U.S. locales this is '.'. 257) In common U.S. locales this is ','.
```

§ 22.4.6.3.2 693

```
string do_grouping() const;
3
         Returns: A pattern defined identically as, but not necessarily equal to, the result of numpunct<chart>::
         do grouping().<sup>258</sup>
   string_type do_curr_symbol() const;
4
         Returns: A string to use as the currency identifier symbol. 259
   string_type do_positive_sign() const;
   string_type do_negative_sign() const;
5
         Returns: do_positive_sign() returns the string to use to indicate a positive monetary value; 260
         do_negative_sign() returns the string to use to indicate a negative value.
   int do_frac_digits() const;
6
         Returns: The number of digits after the decimal radix separator, if any.<sup>261</sup>
   pattern do_pos_format() const;
   pattern do_neg_format() const;
7
         Returns: The specializations required in Table 82 (22.3.1.1.1), namely moneypunct<char>, moneypunct<
         wchar_t>, moneypunct<char,true>, and moneypunct<wchar_t,true>, return an object of type pattern
         initialized to { symbol, sign, none, value }.^{262}
                                                                              [locale.moneypunct.byname]
   22.4.6.4
              Class template moneypunct_byname
     namespace std {
       template <class charT, bool Intl = false>
       class moneypunct_byname : public moneypunct<charT, Intl> {
         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();
       };
     }
             The message retrieval category
                                                                                       [category.messages]
  Class messages<charT> implements retrieval of strings from message catalogs.
             Class template messages
                                                                                            [locale.messages]
   22.4.7.1
   258) To specify grouping by 3s, the value is "003"\ not "3".
   259) For international specializations (second template parameter true) this is typically four characters long, usually three
   letters and a space.
   260) This is usually the empty string.
   261) In common U.S. locales, this is 2.
   262) Note that the international symbol returned by do_curr_sym() usually contains a space, itself; for example, "USD ".
```

§ 22.4.7.1 694

```
namespace std {
       class messages_base {
      public:
         typedef int catalog;
      template <class charT>
       class messages : public locale::facet, public messages_base {
       public:
         typedef charT char_type;
         typedef basic_string<charT> string_type;
         explicit messages(size_t refs = 0);
         catalog open(const basic_string<char>& fn, const locale&) const;
         string_type get(catalog c, int set, int msgid,
                          const string_type& dfault) const;
         void close(catalog c) const;
         static locale::id id;
       protected:
         ~messages();
         virtual catalog do_open(const basic_string<char>&, const locale&) const;
         virtual string_type do_get(catalog, int set, int msgid,
                                    const string_type& dfault) const;
         virtual void do_close(catalog) const;
      };
     }
  Values of type messages_base::catalog usable as arguments to members get and close can be obtained
   only by calling member open.
   22.4.7.1.1 messages members
                                                                           [locale.messages.members]
   catalog open(const basic_string<char>& name, const locale& loc) const;
1
        Returns: do_open(name, loc).
   string_type get(catalog cat, int set, int msgid,
                   const string_type& dfault) const;
        Returns: do_get(cat, set, msgid, dfault).
   void close(catalog cat) const;
        Effects: Calls do_close(cat).
   22.4.7.1.2 messages virtual functions
                                                                             [locale.messages.virtuals]
   catalog do_open(const basic_string<char>& name,
                   const locale& loc) const;
1
        Returns: A value that may be passed to get () to retrieve a message from the message catalog identified
        by the string name according to an implementation-defined mapping. The result can be used until it
        is passed to close().
```

§ 22.4.7.1.2 695

- 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.

- 4 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 implementationdefined mapping. If no such message can be found, returns dfault.

```
void do_close(catalog cat) const;
```

- Requires: cat shall be a catalog obtained from open() and not yet closed.
- 7 Effects: Releases unspecified resources associated with cat.
- 8 Remarks: The limit on such resources, if any, is implementation-defined.

#### 22.4.7.2 Class template messages\_byname

[locale.messages.byname]

```
namespace std {
  template <class charT>
  class messages_byname : public messages<charT> {
  public:
    typedef messages_base::catalog catalog;
    typedef basic_string<charT>    string_type;

    explicit messages_byname(const char*, size_t refs = 0);
    explicit messages_byname(const string&, size_t refs = 0);
  protected:
    ~messages_byname();
  };
}
```

#### 22.4.8 Program-defined facets

[facets.examples]

- 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.
- $^{2}$  [Note: The locale member function templates verify its type and storage class. end note]
- <sup>3</sup> [Example: Traditional global localization is still easy:

§ 22.4.8

```
return MyObject(argc, argv).doit();
    - end example]
<sup>4</sup> [Example: Greater flexibility is possible:
     #include <iostream>
     #include <locale>
     int main() {
       using namespace std;
                                       // the user's preferred locale
       cin.imbue(locale(""));
       cout.imbue(locale::classic());
       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.
6 [Example: Here is an example of the use of locales in a library interface.
     // file: Date.h
     #include <iosfwd>
     #include <string>
     #include <locale>
     class Date {
     public:
       Date(unsigned day, unsigned month, unsigned year);
       std::string asString(const std::locale& = std::locale());
     };
     std::istream& operator>>(std::istream& s, Date& d);
     std::ostream& operator<<(std::ostream& s, Date d);</pre>
<sup>7</sup> This example illustrates two architectural uses of class locale.
8 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.
     // file: Date.C
     #include "Date"
                                       // includes <ctime>
     #include <sstream>
     std::string Date::asString(const std::locale& 1) {
       using namespace std;
       ostringstream s; s.imbue(1);
       s << *this; return s.str();</pre>
     std::istream& operator>>(std::istream& s, Date& d) {
       using namespace std;
       istream::sentry cerberos(s);
       if (cerberos) {
         ios_base::iostate err = goodbit;
```

§ 22.4.8

struct tm t;

```
use_facet< time_get<char> >(s.getloc()).get_date(s, 0, s, err, &t);
          if (!err) d = Date(t.tm_day, t.tm_mon + 1, t.tm_year + 1900);
          s.setstate(err);
        }
        return s;
    — end example]
<sup>10</sup> 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:
      // file: <jctype>
      #include <locale>
      namespace My {
        using namespace std;
        class JCtype : public locale::facet {
        public:
                                        // required for use as a new locale facet
          static locale::id id;
          bool is_kanji (wchar_t c) const;
          JCtype() { }
        protected:
          ~JCtype() { }
      }
      // file: filt.C
      #include <iostream>
      #include <locale>
      #include "jctype"
                                        // above
      \mathtt{std}::locale::id My::JCtype::id; // the static JCtype member declared above.
      int main() {
        using namespace std;
        typedef ctype<wchar_t> wctype;
                                        // the user's preferred locale ...
        locale loc(locale(""),
                                      // and a new feature ...
               new My::JCtype);
        wchar_t c = use_facet<wctype>(loc).widen('!');
        if (!use_facet<My::JCtype>(loc).is_kanji(c))
          cout << "no it isn't!" << endl;</pre>
        return 0;
   The new facet is used exactly like the built-in facets. — end example]
   [Example: Replacing an existing facet is even easier. The code does not define a member id because it is
    reusing the numpunct<charT> facet interface:
      // file: my_bool.C
      #include <iostream>
      #include <locale>
      #include <string>
      namespace My {
        using namespace std;
        typedef numpunct_byname<char> cnumpunct;
        class BoolNames : public cnumpunct {
                                                                                                           698
    § 22.4.8
```

```
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;</pre>
   return 0;
— end example]
```

#### 22.5 Standard code conversion facets

[locale.stdcvt]

The header <codecvt> provides code conversion facets for various character encodings.

Header <codecvt> synopsis

```
namespace std {
  enum codecvt_mode {
    consume_header = 4,
    generate_header = 2,
    little_endian = 1
  };
 template<class Elem, unsigned long Maxcode = 0x10ffff,</pre>
    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.

§ 22.5

— Maxcode is the largest wide-character code that the facet will read or write without reporting a conversion error.

- If (Mode & consume\_header), the facet shall consume an initial header sequence, if present, when reading a multibyte sequence to determine the endianness of the subsequent multibyte sequence to be read.
- If (Mode & generate\_header), the facet shall generate an initial header sequence when writing a multibyte sequence to advertise the endianness of the subsequent multibyte sequence to be written.
- If (Mode & little\_endian), the facet shall generate a multibyte sequence in little-endian order, as opposed to the default big-endian order.

#### 4 For the facet codecvt\_utf8:

- The facet shall convert between UTF-8 multibyte sequences and UCS2 or UCS4 (depending on the size of 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.

#### <sup>5</sup> 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.

### <sup>6</sup> 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

[c.locales]

- Table 93 describes header <clocale>.
- The contents are the same as the Standard C library header <locale.h>.
- <sup>3</sup> Calls to the function setlocale may introduce a data race (17.6.5.9) with other calls to setlocale or with calls to the functions listed in Table 94.

See also: ISO C Clause 7.4.

§ 22.6 700

Table 93 — Header <clocale> synopsis

Type	Name(s)		
Macros:	LC_ALL	LC_COLLATE	LC_CTYPE
	LC_MONETARY	LC_NUMERIC	LC_TIME
	NULL		
Struct:	lconv		
Functions:	localeconv	setlocale	

Table 94 — Potential setlocale data races

fprintf	isprint	iswdigit	localeconv	tolower
fscanf	ispunct	iswgraph	mblen	toupper
isalnum	isspace	iswlower	mbstowcs	towlower
isalpha	isupper	iswprint	mbtowc	towupper
isblank	iswalnum	iswpunct	setlocale	wcscoll
iscntrl	iswalpha	iswspace	strcoll	wcstod
isdigit	iswblank	iswupper	strerror	wcstombs
isgraph	iswcntrl	iswxdigit	strtod	wcsxfrm
islower	iswctype	isxdigit	strxfrm	wctomb

§ 22.6 701

# 23 Containers library

# [containers]

#### 23.1 General

[containers.general]

- This Clause describes components that C++ programs may use to organize collections of information.
- <sup>2</sup> The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 95.

	${f Subclause}$	Header(s)
23.2	Requirements	
23.3	Sequence containers	<array></array>
		<deque></deque>
		<forward_list></forward_list>
		<li>t&gt;</li>
		<vector></vector>
23.4	Associative containers	<map></map>
		<set></set>
23.5	Unordered associative containers	<pre><unordered_map></unordered_map></pre>
		<pre><unordered_set></unordered_set></pre>
23.6	Container adaptors	<queue></queue>
		<stack></stack>

#### 23.2 Container requirements

[container.requirements]

### 23.2.1 General container requirements

[container.requirements.general]

- 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]
- <sup>3</sup> 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 (20.6.8.2). These functions are called only for the container's element type, not for internal types used by the container. [Note: This means, for example, that a node-based container might need to construct nodes containing aligned buffers and call construct to place the element into the buffer. —end note]
- In Tables 96 and 97, 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.

 $\odot$  ISO/IEC N3337

Table 96 — Container requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
X::value type	T		Requires: T is Destructible	compile time
X::reference	lvalue of T			compile time
X::const reference	const lvalue of			compile time
X::iterator	iterator type whose value type is T		any iterator category that meets the forward iterator requirements. convertible to X::const_iterator.	compile time
X::const iterator	constant iterator type whose value type is T		any iterator category that meets the forward iterator requirements.	compile time
X::dif- ference_type	signed integer type		is identical to the difference type of X::iterator and X::const_iterator	compile time
X::size_type	unsigned integer type		size_type can represent any non-negative value of difference_type	compile time
X u;			post: u.empty()	constant
X()			post: X().empty()	constant
X(a)			Requires:T is CopyInsertable into X (see below). post: a == X(a).	linear
X u(a) X u = a;			Requires:T is CopyInsertable into X (see below). post: u == a	linear
X u(rv) X u = rv			post: u shall be equal to the value that rv had before this construction	(Note B)
a = rv	X&	All existing elements of a are either move assigned to or destroyed	a shall be equal to the value that rv had before this assignment	linear
(&a)->~X()	void		note: the destructor is applied to every element of a; all the memory is deallocated.	linear

Table 96 — Container requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
- 1 ()	*++	Semantics	pre-/ post-condition	
a.begin()	iterator;			constant
	const			
	iterator for			
	constant a			
a.end()	iterator;			constant
	const			
	iterator for			
	constant ${\tt a}$			
a.cbegin()	const	const_cast <x< td=""><td></td><td>constant</td></x<>		constant
	iterator	<pre>const&amp;&gt;(a).begin();</pre>		
a.cend()	const	const_cast <x< td=""><td></td><td>constant</td></x<>		constant
	iterator	<pre>const&amp;&gt;(a).end();</pre>		
a == b	convertible to	== is an equivalence	Requires: T is	linear
	bool	relation.	EqualityComparable	
		<pre>distance(a.begin(),</pre>	1 7 1	
		a.end()) ==		
		<pre>distance(b.begin(),</pre>		
		b.end()) &&		
		equal(a.begin(),		
		a.end(),		
		b.begin())		
a != b	convertible to	Equivalent to: !(a		linear
a :- b	bool	== b)		micai
(h)	void	b)	exchanges the	(Note A)
a.swap(b)	VOIG		contents of a and b	(Note A)
( 1)		(1.)	contents of a and b	(NT / A)
swap(a, b)	void	a.swap(b)		(Note A)
r = a	X&		post: r == a.	linear
a.size()	size_type	<pre>distance(a.begin(),</pre>		constant
		a.end())		
a.max_size()	size_type	<pre>distance(begin(),</pre>		constant
		<pre>end()) for the</pre>		
		largest possible		
		container		
a.empty()	convertible to	a.begin() ==		constant
	bool	a.end()		

Notes: the algorithm equal() is defined in Clause 25. Those entries marked "(Note A)" or "(Note B)" have linear complexity for array and have constant complexity for all other standard containers.

<sup>&</sup>lt;sup>5</sup> 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.

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();

<sup>7</sup> Unless otherwise specified, all containers defined in this clause obtain memory using an allocator (see 17.6.3.5). Copy constructors for these container types obtain an allocator by calling allocator\_traits<allocator\_type>::select\_on\_container\_copy\_construction on their first parameters. Move constructors obtain an allocator by move construction from the allocator belonging to the container being moved. Such move

construction of the allocator shall not exit via an exception. All other constructors for these container types take an Allocator& argument (17.6.3.5), 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.

- 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 unqual-ified 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.
- <sup>9</sup> 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 97.

Expression	Return type	Assertion/note	Complexity
		pre-/post-condition	
X::reverse	iterator type whose value type	reverse_iterator <iterator></iterator>	compile time
iterator	is T		
X::const	iterator type whose value type	reverse_iterator <const< td=""><td>compile time</td></const<>	compile time
reverse	is const T	iterator>	
iterator			
a.rbegin()	reverse_iterator;	reverse_iterator(end())	constant
	${\tt const\_reverse\_iterator}\ { m for}$		
	constant a		
a.rend()	reverse_iterator;	<pre>reverse_iterator(begin())</pre>	constant
	${\tt const\_reverse\_iterator}\ { m for}$		
	constant a		
a.crbegin();	const_reverse_iterator	const_cast <x< td=""><td>constant</td></x<>	constant
		<pre>const&amp;&gt;(a).rbegin();</pre>	
a.crend();	const_reverse_iterator	const_cast <x< td=""><td>constant</td></x<>	constant
		<pre>const&amp;&gt;(a).rend();</pre>	

Table 97 — Reversible container requirements

<sup>&</sup>lt;sup>10</sup> Unless otherwise specified (see 23.2.4.1, 23.2.5.1, 23.3.3.4, and 23.3.6.5) all container types defined in this Clause meet the following additional requirements:

— if an exception is thrown by an insert() or emplace() 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]
- 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.
- Table 98 lists operations that are provided for some types of containers but not others. Those containers for which the listed operations are provided shall implement the semantics described in Table 98 unless otherwise stated.

Expression	Return type	Operational semantics	Assertion/note	Complexity
		semantics	pre-/post-condition	
a < b	convertible to bool	<pre>lexicographical compare( a.begin(), a.end(), b.begin(),</pre>	pre: < is defined for values of T. < is a total ordering relationship.	linear
		b.end())		
a > b	convertible to	b < a		linear
	bool			
a <= b	convertible to	!(a > b)		linear
	bool			
a >= b	convertible to bool	!(a < b)		linear

Table 98 — Optional container operations

Note: the algorithm lexicographical\_compare() is defined in Clause 25.

All of the containers defined in this Clause and in (21.4) except array meet the additional requirements of an allocator-aware container, as described in Table 99.

Given a container type X having an allocator\_type identical to A and a value\_type identical to T and given an lvalue m of type A, a pointer P of type T, an expression P of type T, and an rvalue T of type T, the following terms are defined. (If T is not allocator-aware, the terms below are defined as if T were std::allocator<T.)

- T is CopyInsertable into X means that the following expression is well-formed: allocator\_traits<A>::construct(m, p, v);
- T is MoveInsertable into X means that the following expression is well-formed:

allocator\_traits<A>::construct(m, p, rv);

— T is *EmplaceConstructible into X from args*, for zero or more arguments args, means that the following expression is well-formed:

allocator\_traits<A>::construct(m, p, args);

[Note: A container calls allocator\_traits<A>::construct(m, p, args) to construct an element at p using args. The default construct in std::allocator will call::new((void\*)p) T(args), but specialized allocators may choose a different definition. —end note]

In Table 99, 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 99 — Allocator-aware container requirements

Expression		Return type	$\begin{array}{c} \textbf{Assertion/note} \\ \textbf{pre-/post-condition} \end{array}$	Complexity
allocator type	A		Requires:allocator type::value_type is the same as X::value_type.	compile time
<pre>get allocator()</pre>	A			constant
X() X u;			Requires: A is DefaultConstructible. post: u.empty() returns true, u.get_allocator() == A()	constant
X(m) X u(m);			<pre>post: u.empty() returns true, u.get_allocator() == m</pre>	constant
X(t, m) X u(t, m);			<pre>Requires: T is CopyInsertable into X. post: u == t, get_allocator() == m</pre>	linear
X(rv) X u(rv)			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.	constant
X(rv, m) X u(rv, m);			Requires: T is MoveInsertable into X. post: u shall have the same elements, or copies of the elements, that rv had before this construction, get_allocator() == m	<pre>constant if m == rv.get allocator(), otherwise linear</pre>
a = t	X&		Requires: T is CopyInsertable into X and CopyAssignable. post: a == t	linear

Expression	Return type	$\begin{array}{c} \textbf{Assertion/note} \\ \textbf{pre-/post-condition} \end{array}$	Complexity
a = rv	X&	Requires: If allocator traits <allocator_type> ::propagate_on_container move_assignment::value is false, T is MoveInsertable into X and MoveAssignable. All existing elements of a are either move assigned to or destroyed. post: a shall be equal to the value that rv had before this assignment.</allocator_type>	linear
a.swap(b)	void	exchanges the contents of <b>a</b> and <b>b</b>	constant

Table 99 — Allocator-aware container requirements (continued)

#### 23.2.2 Container data races

## [container.requirements.dataraces]

- <sup>1</sup> For purposes of avoiding data races (17.6.5.9), implementations shall consider the following functions to be const: begin, end, rbegin, rend, front, back, data, find, lower\_bound, upper\_bound, equal\_range, at and, except in associative or unordered associative containers, operator[].
- Notwithstanding (17.6.5.9), implementations are required to avoid data races when the contents of the contained object in different elements in the same sequence, excepting vector<br/>
  bool>, are modified concurrently.
- [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

## [sequence.reqmts]

- 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).
- <sup>2</sup> 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.
- In Tables 100 and 101, X denotes a sequence container class, a denotes a value of X containing elements of type T, A denotes X::allocator\_type if 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&&.

<sup>4</sup> The complexities of the expressions are sequence dependent.

Table 100 — Sequence container requirements (in addition to container)

Expression	Return type	Assertion/note
		pre-/post-condition
X(n, t)		Requires: T shall be CopyInsertable into X.
X a(n, t)		<pre>post: distance(begin(), end()) == n</pre>
		Constructs a sequence container with n copies
		of t
X(i, j)		Requires: T shall be EmplaceConstructible
X a(i, j)		into X from *i. For vector, if the iterator
		does not meet the forward iterator
		requirements $(24.2.5)$ , T shall also be
		MoveInsertable into X. Each iterator in the
		range [i,j) shall be dereferenced exactly
		once.
		<pre>post: distance(begin(), end()) ==</pre>
		<pre>distance(i, j)</pre>
		Constructs a sequence container equal to the
		range [i, j)
X(i1);		Equivalent to X(il.begin(), il.end())
a = il;	X&	Requires: T is CopyInsertable into X and
		CopyAssignable. Assigns the range
		<pre>[il.begin(),il.end()) into a. All existing</pre>
		elements of a are either assigned to or
		destroyed.
		Returns: *this.
a.emplace(p, args);	iterator	Requires: T is EmplaceConstructible into X
		from args. For vector and deque, T is also
		MoveInsertable into X and MoveAssignable.
		Effects: Inserts an object of type T
		constructed with
		std::forward <args>(args) before p.</args>
a.insert(p,t)	iterator	Requires: T shall be CopyInsertable into X.
		For vector and deque, T shall also be
		CopyAssignable.
		Effects: Inserts a copy of t before p.
a.insert(p,rv)	iterator	Requires: T shall be MoveInsertable into X.
		For vector and deque, T shall also be
		MoveAssignable.
		Effects: Inserts a copy of rv before p.
a.insert(p,n,t)	iterator	Requires:T shall be CopyInsertable into X
		and CopyAssignable.
		Inserts n copies of t before p.

Table 100 — Sequence container requirements (in addition to container) (continued)

a.insert(p,i,j)  iterator  Requires:T shall be EmplaceConstructible into X from *i. For vector, if the iterator does not meet the forward iterator requirements (24.2.5), T shall also be MoveInsertable into X and MoveAssignab 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 a.insert(p, il); iterator a.insert(p, il.begin(), il.end()).  Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the element pointed to by q  Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the elements in the range [c]	Expression	Return type	Assertion/note
into X from *i. For vector, if the iterator does not meet the forward iterator requirements (24.2.5), T shall also be  MoveInsertable into X and MoveAssignab  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  a.insert(p, il); iterator a.insert(p, il.begin(), il.end()).  a.erase(q) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the element pointed to by q  a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the elements in the range [c]			pre-/post-condition
does not meet the forward iterator requirements (24.2.5), T shall also be MoveInsertable into X and MoveAssignab 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 a.insert(p, il); iterator a.insert(p, il.begin(), il.end()). a.erase(q) iterator Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the element pointed to by q a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the elements in the range [c]	a.insert(p,i,j)	iterator	Requires: T shall be EmplaceConstructible
requirements (24.2.5), T shall also be MoveInsertable into X and MoveAssignab 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 a.insert(p, il); iterator a.insert(p, il.begin(), il.end()). a.erase(q) iterator Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the element pointed to by q a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the elements in the range [c]			
MoveInsertable into X and MoveAssignab  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  a.insert(p, il); iterator a.insert(p, il.begin(), il.end()).  a.erase(q) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the element pointed to by q  a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the elements in the range [c]			
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 a.insert(p, il); iterator a.insert(p, il.begin(), il.end()). a.erase(q) iterator Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the element pointed to by q a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the elements in the range [c]			requirements $(24.2.5)$ , T shall also be
dereferenced exactly once.  pre: i and j are not iterators into a.  Inserts copies of elements in [i, j) before  a.insert(p, il); iterator a.insert(p, il.begin(), il.end()).  a.erase(q) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the element pointed to by q  a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the elements in the range [c]			MoveInsertable into X and MoveAssignable.
pre: i and j are not iterators into a.  Inserts copies of elements in [i, j) before  a.insert(p, il); iterator a.insert(p, il.begin(), il.end()).  a.erase(q) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the element pointed to by q  a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the elements in the range [c]			~ •
Inserts copies of elements in [i, j) before  a.insert(p, il); iterator a.insert(p, il.begin(), il.end()).  a.erase(q) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the element pointed to by q  a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the elements in the range [c]			· ·
a.insert(p, il); iterator a.insert(p, il.begin(), il.end()).  a.erase(q) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the element pointed to by quarters: For vector and deque, T shall be MoveAssignable.  Effects: Erases the elements in the range [content of the content of the cont			
a.erase(q)  iterator  Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the element pointed to by q  a.erase(q1,q2)  iterator  Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the elements in the range [c]			
MoveAssignable.  Effects: Erases the element pointed to by q  a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the elements in the range [c]			
a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the element pointed to by q Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the elements in the range [6]	a.erase(q)	iterator	
a.erase(q1,q2) iterator Requires: For vector and deque, T shall be MoveAssignable.  Effects: Erases the elements in the range [content of the content of the			=
MoveAssignable.  Effects: Erases the elements in the range [c			
Effects: Erases the elements in the range [c	a.erase(q1,q2)	iterator	
q2).			
a.clear() void Destroys all elements in a. Invalidates all	a.clear()	void	v
			references, pointers, and iterators referring to
the elements of a and may invalidate the			
past-the-end iterator.			-
post: a.empty() returns true	(; ;)		
	a.assign(1,j)	vola	Requires: T shall be EmplaceConstructible into X from *i and assignable from *i. For
vector, if the iterator does not meet the			ě
·			forward iterator requirements (24.2.5), T shall
also be MoveInsertable into X.			- \ / /
Each iterator in the range [i,j) shall be			
dereferenced exactly once.			
pre: i, j are not iterators into a.			· ·
			Replaces elements in a with a copy of [i, j).
a.assign(il) void a.assign(il.begin(), il.end()).	a.assign(il)	void	
			Requires: T shall be CopyInsertable into X
and CopyAssignable.	5 , ,		
pre: t is not a reference into a.			
Replaces elements in a with n copies of t.			

<sup>&</sup>lt;sup>5</sup> iterator and const\_iterator types for sequence containers shall be at least of the forward iterator category.

<sup>6</sup> The iterator returned from a.insert(p, t) points to the copy of t inserted into a.

<sup>&</sup>lt;sup>7</sup> The iterator returned from a.insert(p, rv) points to the copy of rv inserted into a.

<sup>&</sup>lt;sup>8</sup> The iterator returned from a.insert(p, n, t) points to the copy of the first element inserted into a, or p if n == 0.

The iterator returned from a.insert(p, i, j) points to the copy of the first element inserted into a, or p if i == j.

The iterator returned from a.insert(p, i1) points to the copy of the first element inserted into a, or p if i1 is empty.

- 11 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.
- <sup>14</sup> 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 shall not participate in overload resolution.

— If the member functions of the forms:

are called with a type InputIterator that does not qualify as an input iterator, then these functions shall not participate in overload resolution.

- The extent to which an implementation determines that a type cannot be an input iterator is unspecified, except that as a minimum integral types shall not qualify as input iterators.
- Table 101 lists operations that are provided for some types of sequence containers but not others. An implementation shall provide these operations for all container types shown in the "container" column, and shall implement them so as to take amortized constant time.

Table 101 -	<ul> <li>Ontional</li> </ul>	seguence	container	operations
Table 101 -	– Орионаі	sequence	contamer	operations

Expression	Return type	Operational semantics	Container
a.front()	reference; const_reference	*a.begin()	basic
	for constant a		string,
			array, deque,
			forward
			list, list,
			vector
a.back()	reference; const_reference	{ auto tmp = a.end();	basic
	for constant a	tmp;	string,
		return *tmp; }	array, deque,
			list, vector

Table 101 — Optional sequence container operations (continued)

Expression	Return type	Operational semantics	Container
a.emplace front(args)	void	Prepends an object of type T constructed with std::forward <args>(args)</args>	<pre>deque, forward list, list</pre>
		Requires: T shall be EmplaceConstructible into X from args.	
a.emplace back(args)	void	Appends an object of type T constructed with std::forward <args>(args)</args>	deque, list, vector
		Requires: T shall be EmplaceConstructible into X from args. For vector, T shall also be MoveInsertable into X.	
a.push front(t)	void	Prepends a copy of t.  Requires: T shall be CopyInsertable into X.	<pre>deque, forward list, list</pre>
a.push front(rv)	void	Prepends a copy of rv.  Requires: T shall be  MoveInsertable into X.	<pre>deque, forward list, list</pre>
a.push back(t)	void	Appends a copy of t.  Requires: T shall be CopyInsertable into X.	basic string, deque, list, vector
a.push back(rv)	void	Appends a copy of rv.  Requires: T shall be  MoveInsertable into X.	basic string, deque, list, vector
a.pop front()	void	Destroys the first element.  Requires: a.empty() shall be false.	<pre>deque, forward list, list</pre>
a.pop_back()	void	Destroys the last element.  Requires: a.empty() shall be false.	basic string, deque, list, vector
a[n]	reference; const_reference for constant a	*(a.begin() + n)	basic string, array, deque, vector
a.at(n)	reference; const_reference for constant a	*(a.begin() + n)	basic string, array, deque, vector

The member function at() provides bounds-checked access to container elements. at() throws out\_of\_-range if n >= a.size().

#### 23.2.4 Associative containers

# [associative.reqmts]

Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: set, multiset, map and multimap.

- <sup>2</sup> 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.
- The phrase "equivalence of keys" means the equivalence relation imposed by the comparison and *not* the operator== on keys. That is, two keys k1 and k2 are considered to be equivalent if for the comparison object comp, comp(k1, k2) == false && comp(k2, k1) == false. For any two keys k1 and k2 in the same container, calling comp(k1, k2) shall always return the same value.
- 4 An associative container supports unique keys if it may contain at most one element for each key. Otherwise, it supports equivalent keys. The set and map classes support unique keys; the multiset and multimap classes support equivalent keys. For multiset and multimap, insert, emplace, and erase preserve the relative ordering of equivalent elements.
- <sup>5</sup> For set and multiset the value type is the same as the key type. For map and multimap it is equal to pair<const Key, T>. Keys in an associative container are immutable.
- iterator of an associative container is of the bidirectional iterator category. For associative containers where the value type is the same as the key type, both iterator and const\_iterator are constant iterators. It is unspecified whether or not iterator and const\_iterator are the same type. [Note: iterator and const\_iterator have identical semantics in this case, and iterator is convertible to const\_iterator. Users can avoid violating the One Definition Rule by always using const\_iterator in their function parameter lists.

   end note]
- The associative containers meet all the requirements of Allocator-aware containers (23.2.1), except that for map and multimap, the requirements placed on value\_type in Table 96 apply instead to key\_type and mapped\_type. [Note: For example, in some cases key\_type and mapped\_type are required to be CopyAssignable even though the associated value\_type, pair<const key\_type, mapped\_type>, is not CopyAssignable. end note]
- In Table 102, X denotes an associative container class, a denotes a value of X, a\_uniq denotes a value of X when X supports unique keys, a\_eq denotes a value of X when X supports multiple keys, 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 102 — Associative container requirements (in addition to container)

Expression	Return type	Assertion/note	Complexity
		$\operatorname{pre-/post-condition}$	
X::key_type	Key	Requires: Key is	compile time
		Destructible.	
mapped_type	T	Requires: T is Destructible.	compile time
(map and			
multimap			
only)			

Table 102 — Associative container requirements (in addition to container) (continued)

Expression	Return type	$egin{array}{l} { m Assertion/note} \ { m pre-/post-condition} \end{array}$	Complexity
X::key compare	Compare	defaults to less <key_type></key_type>	compile time
X::value compare	a binary predicate type	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.	compile time
X(c) X a(c);		Requires: key_compare is CopyConstructible. Effects: Constructs an empty container. Uses a copy of c as a comparison object.	constant
X() X a;		Requires: key_compare is DefaultConstructible. Effects: Constructs an empty container. Uses Compare() as a comparison object	constant
X(i,j,c) X a(i,j,c);		Requires: key_compare is CopyConstructible. value_type is EmplaceConstructible into X from *i.  Effects: Constructs an empty container and inserts elements from the range [i, j) into it; uses c as a comparison object.	N log N in general (N has the value distance(i, j)); linear if [i, j) is sorted with value_comp()
X(i,j) X a(i,j);		Requires: key_compare is DefaultConstructible. value_type is EmplaceConstructible into X from *i. Effects: Same as above, but uses Compare() as a comparison object	same as above
X(i1);		Same as X(il.begin(), il.end()).	<pre>same as X(il.begin(), il.end()).</pre>
a = il	X&	Requires: value_type is CopyInsertable into X and CopyAssignable. Effects: Assigns the range [il.begin(),il.end()) into a. All existing elements of a are either assigned to or destroyed.	<pre>NlogN in general (where N has the value il.size() + a.size()); linear if [il.begin(),il.end()) is sorted with value_comp().</pre>

Table 102 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note	Complexity
		pre-/post-condition	
a.key	X::key	returns the comparison object	constant
comp()	compare	out of which a was	
		constructed.	
a.value	X::value	returns an object of	constant
comp()	compare	value_compare constructed	
		out of the comparison object	
a_uniq.	pair <iterator,< td=""><td>Requires: value_type shall be</td><td>logarithmic</td></iterator,<>	Requires: value_type shall be	logarithmic
emplace(args)	bool>	EmplaceConstructible into X	
		from args.	
		Effects: Inserts a value_type	
		object t constructed with	
		std::forward <args>(args)</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.	1 11 1
a_eq.	iterator	Requires: value_type shall be	logarithmic
emplace(args)		EmplaceConstructible into X	
		from args.	
		Effects: Inserts a value_type	
		object t constructed with	
		std::forward <args>(args)</args>	
		and returns the iterator	
		pointing to the newly inserted	
		element. If a range containing	
		elements equivalent to t exists	
		in a_eq, t is inserted at the end of that range.	
a.emplace	iterator	equivalent to a.emplace(	logarithmic in general, but
hint(p,	10014001	std::forward <args>(args))</args>	
args)		Return value is an iterator	element is inserted right
		pointing to the element with	before p
		the key equivalent to the newly	Doloro P
		inserted element. The element	
		is inserted as close as possible	
		to the position just prior to p.	
		to the position Jane prior to p.	

Table 102 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
a_uniq. insert(t)	pair <iterator, bool=""></iterator,>	Requires: If t is a non-const rvalue expression, value_type shall be MoveInsertable into X; otherwise, value_type shall be CopyInsertable into X.  Effects: Inserts t if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the	logarithmic
a eq.insert(t)	iterator	pair points to the element with key equivalent to the key of t.  Requires: If t is a non-const rvalue expression, value_type shall be MoveInsertable into X; otherwise, value_type shall be CopyInsertable into X.  Effects: Inserts t and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to t exists in a_eq, t is inserted at the end of that range.	logarithmic
a.insert(p, t)	iterator	Requires: If t is a non-const rvalue expression, value_type shall be MoveInsertable into X; otherwise, value_type shall be CopyInsertable into X.  Effects: Inserts t if and only if there is no element with key equivalent to the key of t in containers with unique keys; always inserts t in containers with equivalent keys. always returns the iterator pointing to the element with key equivalent to the key of t. t is inserted as close as possible to the position just prior to p.	logarithmic in general, but amortized constant if t is inserted right before p.

Table 102 — Associative container requirements (in addition to container) (continued)

Expression	Return type	$\begin{array}{c} {\bf Assertion/note} \\ {\bf pre-/post\text{-}condition} \end{array}$	Complexity
a.insert(i,j)	void	Requires: value_type shall be EmplaceConstructible into X from *i. pre: i, j are not iterators into a. inserts each element from the range [i,j) if and only if there is no element with key equivalent to the key of that element in containers with unique keys; always inserts that element in containers with equivalent keys.	$N \log(\mathrm{a.size}() + N)$ (N has the value distance(i, j)
a.insert(il)	void	Equivalent to a.insert(il.begin(), il.end()).	
a.erase(k)	size_type	erases all elements in the container with key equivalent to k. returns the number of erased elements.	$\log(\text{a.size}()) + \text{a.count}(k)$
a.erase(q)	iterator	erases the element pointed to by q. Returns an iterator pointing to the element immediately following q prior to the element being erased. If no such element exists, returns a.end().	amortized constant
a.erase( q1, q2)	iterator	erases all the elements in the range [q1,q2). Returns q2.	$\log(\text{a.size}()) + N \text{ where } N$ has the value distance(q1, q2).
a.clear()	void	<pre>a.erase(a.begin(),a.end()) post: a.empty() returns true</pre>	linear in a.size().
a.find(k)	iterator; const iterator for constant a.	returns an iterator pointing to an element with the key equivalent to k, or a.end() if such an element is not found	logarithmic
a.count(k)	size_type	returns the number of elements with key equivalent to $k$	$\log(\text{a.size}()) + \text{a.count}(k)$
a.lower bound(k)	<pre>iterator; const iterator for constant a.</pre>	returns an iterator pointing to the first element with key not less than k, or a.end() if such an element is not found.	logarithmic

Expression	Return type	$\begin{array}{c} {\bf Assertion/note} \\ {\bf pre-/post-condition} \end{array}$	Complexity
a.upper bound(k)	iterator; const iterator for constant a.	returns an iterator pointing to the first element with key greater than k, or a.end() if such an element is not found.	logarithmic
a.equal range(k)	<pre>pair<iterator, iterator="">; pair<const const="" iterator="" iterator,=""> for constant a.</const></iterator,></pre>	<pre>equivalent to make pair(a.lower_bound(k), a.upper_bound(k)).</pre>	logarithmic

Table 102 — Associative container requirements (in addition to container) (continued)

- The insert and emplace members shall not affect the validity of iterators and references to the container, and the erase members shall invalidate only iterators and references to the erased elements.
- The 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,

<sup>11</sup> For associative containers with unique keys the stronger condition holds,

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

#### [associative.regmts.except]

- 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).
- <sup>2</sup> For associative containers, if an exception is thrown by any operation from within an insert or emplace function inserting a single element, the insertion has no effect.
- <sup>3</sup> 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

# [unord.req]

- Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case complexity for most operations is linear, but the average case is much faster. The library provides four unordered associative containers: unordered\_set, unordered\_map, unordered\_multiset, and unordered\_multimap.
- Unordered associative containers conform to the requirements for Containers (23.2), except that the expressions a == b and a != b have different semantics than for the other container types.
- Each unordered associative container is parameterized by Key, by a function object type Hash that meets the Hash requirements (17.6.3.4) and acts as a hash function for argument values of type Key, and by a binary predicate Pred that induces an equivalence relation on values of type Key. Additionally, unordered\_map and unordered\_multimap associate an arbitrary mapped type T with the Key.

<sup>4</sup> A hash function is a function object that takes a single argument of type Key and returns a value of type std::size\_t.

- Two values k1 and k2 of type Key are considered 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]
- 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.
- <sup>7</sup> 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 96 apply instead to key\_type and mapped\_type. [Note: For example, key\_type and mapped\_type are sometimes required to be CopyAssignable even though the associated value\_type, pair<const key\_type, mapped\_type>, is not CopyAssignable. —end note]
- In table 103: X is an unordered associative container class, a is an object of type X, b is a possibly const object of type X, a\_uniq is an object of type X when X supports unique keys, a\_eq is an object of type X when X supports equivalent keys, i and j are input iterators that refer to value\_type, [i, j) is a valid range, p and q2 are valid const iterators to a, q and q1 are valid dereferenceable const iterators to a, [q1, q2) is a valid range in a, il designates an object of type initializer\_list<value\_type>, t is a value of type X::value\_type, k is a value of type key\_type, hf is a possibly const value of type 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 103 — Unordered associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
X::key_type	Key	Requires: Key shall be	compile time
		Destructible.	
X::mapped_type (unordered_map and unordered_multimap only)	Т	Requires: T is Destructible.	compile time
X::hasher	Hash	Hash shall be a unary function object type such that the expression hf(k) has type std::size_t.	compile time

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
X::key_equal	Pred	Pred shall be a binary predicate	compile time
		that takes two arguments of	
		type Key. Pred is an	
		equivalence relation.	
X::local_iterator	An iterator type whose	A local_iterator object may	compile time
	category, value type,	be used to iterate through a	
	difference type, and	single bucket, but may not be	
	pointer and reference	used to iterate across buckets.	
	types are the same as		
	X::iterator's.		
X::const_local	An iterator type whose	${ m A}$ const_local_iterator	compile time
iterator	category, value type,	object may be used to iterate	
	difference type, and	through a single bucket, but	
	pointer and reference	may not be used to iterate	
	types are the same as	across buckets.	
	${\tt X::const\_iterator's}.$		
X(n, hf, eq)	X	Requires: hasher and	$\mathscr{O}(n)$
X a(n, hf, eq)		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.	
X(n, hf)	X	Requires: hasher is	$\mathscr{O}(n)$
X a(n, hf)		CopyConstructible and	
		key_equal is	
		DefaultConstructible.	
		Effects: Constructs an empty	
		container with at least n	
		buckets, using hf as the hash	
		function and key_equal() as	
		the key equality predicate.	
X(n)	X	Requires: hasher and	$\mathscr{O}(n)$
X a(n)		key_equal are	
		DefaultConstructible.	
		Effects: Constructs an empty	
		container with at least n	
		buckets, using hasher() as the	
		hash function and key_equal()	
		as the key equality predicate.	

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
X() X a	X	Requires: hasher and key_equal are DefaultConstructible. Effects: Constructs an empty container with an unspecified number of buckets, using hasher() as the hash function and key_equal() as the key	constant
X(i, j, n, hf, eq) X a(i, j, n, hf, eq)	X	equality predicate.  Requires: hasher and key_equal are CopyConstructible. value_type is EmplaceConstructible into X from *i.  Effects: Constructs an empty container with at least n buckets, using hf as the hash function and eq as the key equality predicate, and inserts elements from [i, j) into it.	Average case $\mathcal{O}(N)$ ( $N$ is distance(i, j)), worst case $\mathcal{O}(N^2)$
X(i, j, n, hf) X a(i, j, n, hf)	X	Requires: hasher is CopyConstructible and key_equal is DefaultConstructible. value_type is EmplaceConstructible into X from *i. Effects: Constructs an empty container with at least n buckets, using hf as the hash function and key_equal() as the key equality predicate, and inserts elements from [i, j) into it.	Average case $\mathcal{O}(N)$ ( $N$ is distance(i, j)), worst case $\mathcal{O}(N^2)$

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression		Return type	Assertion/note pre-/post-condition	Complexity
X(i, j, n) X a(i, j, n)	X		Requires: hasher and key_equal are DefaultConstructible. value_type is EmplaceConstructible into X from *i. Effects: Constructs an empty container with at least n buckets, using hasher() as the hash function and key_equal() as the key equality predicate, and inserts elements from [i, j) into it.	Average case $\mathcal{O}(N)$ ( $N$ is distance(i, j)), worst case $\mathcal{O}(N^2)$
X(i, j) X a(i, j)	X		Requires: hasher and key_equal are DefaultConstructible. value_type is EmplaceConstructible into X from *i.  Effects: Constructs an empty container with an unspecified number of buckets, using hasher() as the hash function and key_equal() as the key equality predicate, and inserts elements from [i, j) into it.	Average case $\mathcal{O}(N)$ ( $N$ is distance(i, j)), worst case $\mathcal{O}(N^2)$
X(il)	Х		Same as X(il.begin(), il.end()).	Same as X(il.begin(), il.end()).
X(b) X a(b)	X		Copy constructor. In addition to the requirements of Table 96, copies the hash function, predicate, and maximum load factor.	Average case linear in b.size(), worst case quadratic.
a = b	X&		Copy assignment operator. In addition to the requirements of Table 96, copies the hash function, predicate, and maximum load factor.	Average case linear in b.size(), worst case quadratic.
a = il	X&		Requires: value_type is CopyInsertable into X and CopyAssignable. Effects: Assigns the range [il.begin(),il.end()) into a. All existing elements of a are either assigned to or destroyed.	Same as a = X(i1).

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
b.hash_function()	hasher	Returns b's hash function.	constant
b.key_eq()	key_equal	Returns b's key equality predicate.	constant
a_uniq. emplace(args)	pair <iterator, bool=""></iterator,>	Requires: value_type shall be EmplaceConstructible into X from args.  Effects: Inserts a value_type object t constructed with std::forward <args>(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.</args>	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(\mathtt{a\_uniq}.\mathtt{size}())$ .
a_eq.emplace(args)	iterator	Requires: value_type shall be EmplaceConstructible into X from args.  Effects: Inserts a value_type object t constructed with std::forward <args>(args) and returns the iterator pointing to the newly inserted element.</args>	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(a_eq.$ size()).
a.emplace_hint(p, args)	iterator	Requires: value_type shall be EmplaceConstructible into X from args.  Effects: Equivalent to a.emplace( 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.</args>	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(\mathtt{a.size}())$ .

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
a_uniq.insert(t)	pair <iterator, bool&gt;</iterator, 	Requires: If t is a non-const rvalue expression, value_type shall be MoveInsertable into X; otherwise, value_type shall be CopyInsertable into X.  Effects: Inserts t if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair indicates whether the insertion takes place, and the iterator component points to the element with key equivalent to the key of t.	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(a\_uniq.size())$ .
a_eq.insert(t)	iterator	Requires: If t is a non-const rvalue expression, value_type shall be MoveInsertable into X; otherwise, value_type shall be CopyInsertable into X.  Effects: Inserts t, and returns an iterator pointing to the newly inserted element.	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(\mathtt{a_eq}$ . size()).
a.insert(q, t)	iterator	Requires: If t is a non-const rvalue expression, value_type shall be MoveInsertable into X; otherwise, value_type shall be CopyInsertable into X.  Effects: 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.	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(a.size())$ .
a.insert(i, j)	void	Requires: value_type shall be EmplaceConstructible into X from *i.  Pre: i and j are not iterators in a. Equivalent to a.insert(t) for each element in [i,j).	Average case $\mathcal{O}(N)$ , where $N$ is distance(i, j). Worst case $\mathcal{O}(N * (a.size()) + N)$ .

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
a.insert(il)	void	Same as	Same as
		<pre>a.insert(il.begin(),</pre>	a.insert(
		il.end()).	<pre>il.begin(),</pre>
			<pre>il.end()).</pre>
a.erase(k)	size_type	Erases all elements with key	Average case
		equivalent to k. Returns the	$\mathcal{O}(\mathtt{a.count(k)}).$
		number of elements erased.	Worst case
			$\mathscr{O}(\mathtt{a.size()}).$
a.erase(q)	iterator	Erases the element pointed to	Average case
		by q. Return value is the	$\mathcal{O}(1)$ , worst
		iterator immediately following $q$	case
		prior to the erasure.	$\mathscr{O}(\mathtt{a.size()}).$
a.erase(q1, q2)	iterator	Erases all elements in the range	Average case
		[q1, q2). Return value is the	linear in
		iterator immediately following	<pre>distance(q1,</pre>
		the erased elements prior to the	q2), worst case
		erasure.	$\mathscr{O}(\mathtt{a.size()}).$
a.clear()	void	Erases all elements in the	Linear.
		container. Post: a.empty()	
		returns true	
b.find(k)	iterator;	Returns an iterator pointing to	Average case
	${\tt const\_iterator} \ {\tt for}$	an element with key equivalent	$\mathcal{O}(1)$ , worst
	const b.	to k, or b.end() if no such	case
		element exists.	$\mathscr{O}(\mathtt{b.size()}).$
b.count(k)	size_type	Returns the number of elements	Average case
		with key equivalent to k.	$\mathcal{O}(1)$ , worst
			case
			$\mathscr{O}(\mathtt{b.size()}).$
b.equal_range(k)	pair <iterator,< td=""><td>Returns a range containing all</td><td>Average case</td></iterator,<>	Returns a range containing all	Average case
	<pre>iterator&gt;;</pre>	elements with keys equivalent	$\mathcal{O}(\texttt{b.count(k)}).$
	pair <const< td=""><td>to k. Returns</td><td>Worst case</td></const<>	to k. Returns	Worst case
	iterator,	<pre>make_pair(b.end(),</pre>	$\mathscr{O}(\mathtt{b.size()}).$
	<pre>const_iterator&gt; for</pre>	b.end()) if no such elements	
	const b.	exist.	
b.bucket_count()	size_type	Returns the number of buckets	Constant
		that b contains.	
b.max_bucket	size_type	Returns an upper bound on the	Constant
count()		number of buckets that <b>b</b> might	
		ever contain.	

Table 103 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
b.bucket(k)	size_type	Pre: b.bucket_count() > 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()).	Constant
b.bucket_size(n)	size_type	Pre: n shall be in the range [0, b.bucket_count()). Returns the number of elements in the n th bucket.	$\mathscr{O}(\texttt{b.bucket}\$ $\texttt{size}(\texttt{n}))$
b.begin(n)	<pre>local_iterator; const_local iterator for const b.</pre>	<pre>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).</pre>	Constant
b.end(n)	<pre>local_iterator; const_local iterator for const b.</pre>	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.	Constant
b.cbegin(n)	const_local iterator	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.	Constant
b.cend(n)	const_local iterator	Pre: n shall be in the range [0, b.bucket_count()).	Constant
b.load_factor()	float	Returns the average number of elements per bucket.	Constant
b.max_load_factor()	float	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.	Constant
a.max_load factor(z)	void	Pre: <b>z</b> shall be positive. May change the container's maximum load load factor, using <b>z</b> as a hint.	Constant

Expression	Return type	Assertion/note pre-/post-condition	Complexity
a.rehash(n)	void	Post: a.bucket_count() >	Average case
		a.size() /	linear in
		$a.max_load_factor()$ and	<pre>a.size(),</pre>
		a.bucket_count() >= n.	worst case
			quadratic.
a.reserve(n)	void	Same as a.rehash(ceil(n /	Average case
		<pre>a.max_load_factor())).</pre>	linear in

Table 103 — Unordered associative container requirements (in addition to container) (continued)

- Two unordered containers a and b compare equal if a.size() == b.size() and, for every equivalentkey 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  $\mathcal{O}(N^2)$ , e.g., for a pathologically bad hash function). The behavior of a program that uses operator == or operator! = on unordered containers is undefined unless the Hash and Pred function objects respectively have the same behavior for both containers and the equality comparison operator for Key is a refinement  $^{263}$  of the partition into equivalent-key groups produced by Pred.
- 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.
- The insert and emplace members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The erase members shall invalidate only iterators and references to the erased elements.
- The insert and emplace members shall not affect the validity of iterators if (N+n) < z \* 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

[unord.req.except]

a.size(), worst case quadratic.

- <sup>1</sup> 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).
- <sup>2</sup> For unordered associative containers, if an exception is thrown by any operation other than the container's hash function from within an insert or emplace function inserting a single element, the insertion has no effect.
- <sup>3</sup> 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).

§ 23.2.5.1 727

<sup>263)</sup> Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.

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

[sequences]

### 23.3.1 In general

[sequences.general]

- The headers <array>, <deque>, <forward\_list>, <list>, and <vector> define template classes that meet the requirements for sequence containers.
- The headers <queue> and <stack> define container adaptors (23.6) that also meet the requirements for sequence containers.

```
Header <array> synopsis
```

```
#include <initializer_list>
namespace std {
  template <class T, size_t N > struct array;
  template <class T, size_t N>
    bool operator==(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator!=(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator<(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator>(const array<T,N>& x, const array<T,N>& y);
 template <class T, size_t N>
    bool operator<=(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N>
    bool operator>=(const array<T,N>& x, const array<T,N>& y);
  template <class T, size_t N >
    void swap(array<T,N>& x, array<T,N>& y) noexcept(noexcept(x.swap(y)));
  template <class T> class tuple_size;
  template <size_t I, class T> class tuple_element;
  template <class T, size_t N>
    struct tuple_size<array<T, N> >;
  template <size_t I, class T, size_t N>
    struct tuple_element<I, array<T, N> >;
  template <size_t I, class T, size_t N>
    T& get(array<T, N>&) noexcept;
  template <size_t I, class T, size_t N>
    T&& get(array<T, N>&&) noexcept;
 template <size_t I, class T, size_t N>
    const T& get(const array<T, N>&) noexcept;
 Header <deque> synopsis
#include <initializer_list>
namespace std {
 template <class T, class Allocator = allocator<T> > class deque;
  template <class T, class Allocator>
    bool operator==(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
```

§ 23.3.1 728

```
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
#include <initializer_list>
namespace std {
  template <class T, class Allocator = allocator<T> > class forward_list;
  template <class T, class Allocator>
    bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator <= (const forward_list < T, Allocator > & x, const forward_list < T, Allocator > & y);
 template <class T, class Allocator>
    void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
 Header <list> synopsis
#include <initializer_list>
namespace std {
  template <class T, class Allocator = allocator<T> > class list;
  template <class T, class Allocator>
    bool operator==(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <class T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>& y);
 Header <vector> synopsis
#include <initializer_list>
```

§ 23.3.1 729

```
namespace std {
  template <class T, class Allocator = allocator<T> > class vector;
  template <class T, class Allocator>
    bool operator==(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <class T, class Allocator>
    void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
  template <class Allocator> class vector<bool, Allocator>;
  // hash support
  template <class T> struct hash;
  template <class Allocator> struct hash<vector<bool, Allocator> >;
```

## 23.3.2 Class template array

array

#### 23.3.2.1 Class template array overview

[array.overview]

- 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<T, N> a = { initializer-list };
```

where initializer-list is a comma-separated list of up to N elements whose types are convertible to T.

An array satisfies all of the requirements of a container and of a reversible container (23.2), except that a default constructed array object is not empty and that swap does not have constant complexity. An array satisfies some of the requirements of a sequence container (23.2.3). Descriptions are provided here only for operations on array that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <class T, size_t N >
  struct array {
    // types:
    typedef T&
                                                   reference;
    typedef const T&
                                                   const_reference;
    typedef implementation-defined
                                                    iterator:
    typedef implementation-defined
                                                    const_iterator;
    typedef size_t
                                                   size_type;
    typedef ptrdiff_t
                                                   difference_type;
                                                   value_type;
    typedef T
    typedef T*
                                                   pointer;
    typedef const T*
                                                   const_pointer;
    typedef reverse_iterator<iterator>
                                                   reverse_iterator;
    typedef reverse_iterator<const_iterator>
                                                   const_reverse_iterator;
```

§ 23.3.2.1 730

```
Т
                                      // exposition only
                 elems[N]:
         // no explicit construct/copy/destroy for aggregate type
         void fill(const T& u);
         void swap(array&) noexcept(noexcept(swap(declval<T&>(), declval<T&>())));
         // iterators:
         iterator
                                 begin() noexcept;
         const_iterator
                                 begin() const noexcept;
         iterator
                                 end() noexcept;
         const_iterator
                                 end() const noexcept;
                                 rbegin() noexcept;
         reverse_iterator
         const_reverse_iterator rbegin() const noexcept;
         reverse_iterator
                                 rend() noexcept;
         const_reverse_iterator rend() const noexcept;
         const_iterator
                                 cbegin() const noexcept;
         const_iterator
                                 cend() const noexcept;
         const_reverse_iterator crbegin() const noexcept;
         const_reverse_iterator crend() const noexcept;
         // capacity:
         constexpr size_type size() noexcept;
         constexpr size_type max_size() noexcept;
         constexpr bool
                              empty() noexcept;
         // 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;
                   data() noexcept;
         const T * data() const noexcept;
       };
<sup>4</sup> [Note: The member variable elems is shown for exposition only, to emphasize that array is a class aggregate.
   The name elems is not part of array's interface. — end note]
   23.3.2.2
             array constructors, copy, and assignment
                                                                                              [array.cons]
  The conditions for an aggregate (8.5.1) shall be met. Class array 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. In addition
   to the requirements specified in the container requirements table, the implicit move constructor and move
   assignment operator for array require that T be MoveConstructible or MoveAssignable, respectively.
   23.3.2.3 array specialized algorithms
                                                                                            [array.special]
   template <class T, size_t N> void swap(array<T,N>& x, array<T,N>& y) noexcept(noexcept(x.swap(y)));
        Effects:
   § 23.3.2.3
                                                                                                        731
```

```
x.swap(y);
         Complexity: linear in N.
   23.3.2.4 array::size
                                                                                                [array.size]
   \label{template lass T, size_t N> constexpr size_type array < T, N> :: size() no except;}
                                                                                               [array.data]
   23.3.2.5 array::data
   T *data() noexcept;
   const T *data() const noexcept;
        Returns: elems.
   23.3.2.6 array::fill
                                                                                                 [array.fill]
   void fill(const T& u);
        Effects: fill_n(begin(), N, u)
   23.3.2.7 array::swap
                                                                                              [array.swap]
   void swap(array& y) noexcept(noexcept(swap(declval<T&>(), declval<T&>())));
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.
                                                                                               [array.zero]
   23.3.2.8 Zero sized arrays
   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.
<sup>3</sup> The effect of calling front() or back() for a zero-sized array is undefined.
4 Member function swap() shall have a noexcept-specification which is equivalent to noexcept(true).
             Tuple interface to class template array
                                                                                              [array.tuple]
   23.3.2.9
   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.
         Value: The type T.
   template <size_t I, class T, size_t N> T& get(array<T, N>& a) noexcept;
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.
```

§ 23.3.2.9 732

```
template <size_t I, class T, size_t N> T&& get(array<T, N>&& a) noexcept;

Effects: Equivalent to return std::move(get<I>(a));

template <size_t I, class T, size_t N> const T& get(const array<T, N>& a) noexcept;

Requires: I < N. The program is ill-formed if I is out of bounds.

Returns: A const reference to the Ith element of a, where indexing is zero-based.
```

## 23.3.3 Class template deque

[deque]

## 23.3.3.1 Class template deque overview

[deque.overview]

- A deque is a sequence container that, like a vector (23.3.6), supports random access iterators. In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take linear time. That is, a deque is especially optimized for pushing and popping elements at the beginning and end. As with vectors, storage management is handled automatically.
- A deque satisfies all of the requirements of a container, of a reversible container (given in tables in 23.2), of a sequence container, including the optional sequence container requirements (23.2.3), and of an allocator-aware container (Table 99). Descriptions are provided here only for operations on deque that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <class T, class Allocator = allocator<T> >
  class deque {
 public:
    // types:
    typedef value_type&
                                                    reference;
    typedef const value_type&
                                                    const reference:
                                                                      // See 23.2
    {\tt typedef} \ \textit{implementation-defined}
                                                    iterator;
    {\tt typedef} \ \textit{implementation-defined}
                                                     const_iterator; // See 23.2
                                                                     // See 23.2
    typedef implementation-defined
                                                     size_type;
    typedef implementation-defined
                                                     difference_type; // See 23.2
    typedef T
                                                    value_type;
    typedef Allocator
                                                    allocator_type;
    typedef typename allocator_traits<Allocator>::pointer
                                                                       pointer;
    typedef typename allocator_traits<Allocator>::const_pointer
                                                                       const_pointer;
    typedef std::reverse_iterator<iterator>
                                                    reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    // 23.3.3.2, construct/copy/destroy:
    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& x);
    deque(deque&&);
    deque(const deque&, const Allocator&);
    deque(deque&&, const Allocator&);
    deque(initializer_list<T>, const Allocator& = Allocator());
    ~deque();
    deque& operator=(const deque& x);
    deque& operator=(deque&& x);
```

§ 23.3.3.1 733

```
deque& operator=(initializer_list<T>);
template <class InputIterator>
  void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
void assign(initializer_list<T>);
allocator_type get_allocator() const noexcept;
// iterators:
iterator
                      begin() noexcept;
const_iterator
                     begin() const noexcept;
iterator
                      end() noexcept;
const_iterator
                      end() const noexcept;
reverse_iterator
                      rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
                       cbegin() const noexcept;
const_iterator
const_iterator
                       cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;
// 23.3.3.3, capacity:
size_type size() const noexcept;
size_type max_size() const noexcept;
         resize(size_type sz);
void
void
         resize(size_type sz, const T& c);
         shrink_to_fit();
void
bool
         empty() const noexcept;
// element access:
reference
               operator[](size_type n);
const_reference operator[](size_type n) const;
reference at(size_type n);
const_reference at(size_type n) const;
              front();
reference
const_reference front() const;
reference back();
const_reference back() const;
// 23.3.3.4, modifiers:
template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
template <class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template <class InputIterator>
  iterator insert (const_iterator position, InputIterator first, InputIterator last);
```

§ 23.3.3.1 734

```
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&);
         void
                  clear() noexcept;
       };
       template <class T, class Allocator>
         bool operator == (const deque < T, Allocator > & x, const deque < T, Allocator > & y);
       template <class T, class Allocator>
         bool operator< (const deque<T,Allocator>& x, const deque<T,Allocator>& y);
       template <class T, class Allocator>
         bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
       template <class T, class Allocator>
         bool operator> (const deque<T,Allocator>& x, const deque<T,Allocator>& y);
       template <class T, class Allocator>
         bool operator>=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
       template <class T, class Allocator>
         bool operator<=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
       // specialized algorithms:
       template <class T, class Allocator>
         void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
   23.3.3.2
             deque constructors, copy, and assignment
                                                                                            [deque.cons]
   explicit deque(const Allocator& = Allocator());
1
        Effects: Constructs an empty deque, using the specified allocator.
2
        Complexity: Constant.
   explicit deque(size_type n);
3
        Effects: Constructs a deque with n value-initialized elements.
4
        Requires: T shall be DefaultConstructible.
5
        Complexity: Linear in n.
   deque(size_type n, const T& value,
         const Allocator& = Allocator());
        Effects: Constructs a deque with n copies of value, using the specified allocator.
7
        Requires: T shall be CopyInsertable into *this.
8
        Complexity: Linear in n.
   template <class InputIterator>
     deque(InputIterator first, InputIterator last,
           const Allocator& = Allocator());
   § 23.3.3.2
                                                                                                       735
```

```
9
         Effects: Constructs a deque equal to the range [first,last), using the specified allocator.
10
         Complexity: distance(first, last).
   template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
11
         Effects:
           erase(begin(), end());
           insert(begin(), first, last);
   void assign(size_type n, const T& t);
12
         Effects:
           erase(begin(), end());
           insert(begin(), n, t);
   23.3.3.3
              deque capacity
                                                                                        [deque.capacity]
   void resize(size_type sz);
         Effects: If sz <= size(), equivalent to erase(begin() + sz, end());. If size() < sz, appends
         sz - size() value-initialized elements to the sequence.
2
         Requires: T shall be DefaultConstructible.
   void resize(size_type sz, const T& c);
3
         Effects:
           if (sz > size())
             insert(end(), sz-size(), c);
           else if (sz < size())
             erase(begin()+sz, end());
           else
                               // do nothing
             ;
4
         Requires: T shall be CopyInsertable into *this.
   void shrink_to_fit();
5
         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
   23.3.3.4
             deque modifiers
                                                                                       [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>);
   § 23.3.3.4
                                                                                                      736
```

```
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);
```

Effects: An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. An insertion at either end of the deque invalidates all the iterators to the deque, but has no effect on the validity of references to elements of the deque.

- Remarks: If an exception is thrown other than by the copy constructor, 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-CopyInsertable T, the effects are unspecified.
- <sup>3</sup> 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);
```

- 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.
- Complexity: The number of calls to the destructor is the same as the number of elements erased, but the number of calls to the assignment operator is no more than the lesser of the number of elements before the erased elements and the number of elements after the erased elements.
- 6 Throws: Nothing unless an exception is thrown by the copy constructor, move constructor, assignment operator, or move assignment operator of T.

#### 23.3.3.5 deque specialized algorithms

[deque.special]

# 23.3.4 Class template forward\_list

[forwardlist]

# 23.3.4.1 Class template forward\_list overview

[forwardlist.overview]

- A forward\_list is a container that supports forward iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Fast random access to list elements is not supported. [Note: It is intended that forward\_list have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted. end note]
- A forward\_list satisfies all of the requirements of a container (Table 96), except that the size() member function is not provided. A forward\_list also satisfies all of the requirements for an allocator-aware container (Table 99). In addition, a forward\_list provides the assign member functions (Table 100) and

§ 23.3.4.1 737

several of the optional container requirements (Table 101). Descriptions are provided here only for operations on forward\_list that are not described in that table or for operations where there is additional semantic information

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 acess 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

```
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
                                                    // See 23.2
    typedef implementation-defined size_type;
    typedef implementation-defined difference_type; // See 23.2
    typedef T value_type;
    typedef Allocator allocator_type;
    typedef typename allocator_traits<Allocator>::pointer
                                                                   pointer;
    typedef typename allocator_traits<Allocator>::const_pointer
                                                                   const_pointer;
    // 23.3.4.2, construct/copy/destroy:
    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();
    forward_list& operator=(const forward_list& x);
    forward_list& operator=(forward_list&& 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 noexcept;
    // 23.3.4.3, iterators:
    iterator before_begin() noexcept;
    const_iterator before_begin() const noexcept;
    iterator begin() noexcept;
    const_iterator begin() const noexcept;
    iterator end() noexcept;
    const_iterator end() const noexcept;
    const_iterator cbegin() const noexcept;
```

§ 23.3.4.1 738

```
const_iterator cbefore_begin() const noexcept;
const_iterator cend() const noexcept;
// capacity:
bool empty() const noexcept;
size_type max_size() const noexcept;
// 23.3.4.4, element access:
reference front();
const_reference front() const;
// 23.3.4.5, modifiers:
template <class... Args> void emplace_front(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void pop_front();
template <class... Args> iterator emplace_after(const_iterator position, Args&&... args);
iterator insert_after(const_iterator position, const T& x);
iterator insert_after(const_iterator position, T&& x);
iterator insert_after(const_iterator position, size_type n, const T& x);
template <class InputIterator>
  iterator insert_after(const_iterator position, InputIterator first, InputIterator last);
iterator insert_after(const_iterator position, initializer_list<T> il);
iterator erase_after(const_iterator position);
iterator erase_after(const_iterator position, const_iterator last);
void swap(forward_list&);
void resize(size_type sz);
void resize(size_type sz, const value_type& c);
void clear() noexcept;
// 23.3.4.6, forward_list operations:
void splice_after(const_iterator position, forward_list& x);
void splice_after(const_iterator position, forward_list&& x);
void splice_after(const_iterator position, forward_list& x,
                  const_iterator i);
void splice_after(const_iterator position, forward_list&& x,
                  const_iterator i);
void splice_after(const_iterator position, forward_list& x,
                  const_iterator first, const_iterator last);
void splice_after(const_iterator position, forward_list&& x,
                  const_iterator first, const_iterator last);
void remove(const T& value);
template <class Predicate> void remove_if(Predicate pred);
void unique();
template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);
void merge(forward_list& x);
void merge(forward_list&& x);
template <class Compare> void merge(forward_list& x, Compare comp);
```

§ 23.3.4.1 739

```
template <class Compare> void merge(forward_list&& x, Compare comp);
         void sort();
         template <class Compare> void sort(Compare comp);
         void reverse() noexcept;
       };
       // Comparison operators
       template <class T, class Allocator>
         bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
       template <class T, class Allocator>
         bool operator< (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
       template <class T, class Allocator>
         bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
       template <class T, class Allocator>
         bool operator> (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
       template <class T, class Allocator>
         bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
       template <class T, class Allocator>
         bool operator<=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
       // 23.3.4.7, specialized algorithms:
       template <class T, class Allocator>
         void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
   23.3.4.2 forward_list constructors, copy, assignment
                                                                                       [forwardlist.cons]
   explicit forward_list(const Allocator& = Allocator());
1
         Effects: Constructs an empty forward_list object using the specified allocator.
2
         Complexity: Constant.
   explicit forward_list(size_type n);
3
         Effects: Constructs a forward_list object with n value-initialized elements.
4
         Requires: T shall be DefaultConstructible.
5
         Complexity: Linear in n.
   forward_list(size_type n, const T& value, const Allocator& = Allocator());
6
         Effects: Constructs a forward_list object with n copies of value using the specified allocator.
7
         Requires: T shall be CopyInsertable into *this.
         Complexity: Linear in n.
   template <class InputIterator>
     forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
9
         Effects: Constructs a forward_list object equal to the range [first,last).
10
         Complexity: Linear in distance(first, last).
```

§ 23.3.4.2 740

```
template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
11
         Effects: clear(); insert_after(before_begin(), first, last);
   void assign(size_type n, const T& t);
12
         Effects: clear(); insert_after(before_begin(), n, t);
   23.3.4.3 forward_list iterators
                                                                                        [forwardlist.iter]
   iterator before_begin() noexcept;
   const_iterator before_begin() const noexcept;
    const_iterator cbefore_begin() const noexcept;
1
         Returns: A non-dereferenceable iterator that, when incremented, is equal to the iterator returned by
         begin().
2
         Effects: cbefore_begin() is equivalent to const_cast<forward_list const&>(*this).before_-
         begin().
3
         Remarks: before_begin() == end() shall equal false.
   23.3.4.4 forward list element access
                                                                                     [forwardlist.access]
   reference front();
    const_reference front() const;
         Returns: *begin()
   23.3.4.5
             forward_list modifiers
                                                                                 [forwardlist.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.
   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.
   void push_front(const T& x);
   void push_front(T&& x);
3
         Effects: Inserts a copy of x at the beginning of the list.
   void pop_front();
         Effects: erase_after(before_begin())
   iterator insert_after(const_iterator position, const T& x);
   iterator insert_after(const_iterator position, T&& x);
```

§ 23.3.4.5 741

```
5
          Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).
6
          Effects: Inserts a copy of x after position.
7
          Returns: An iterator pointing to the copy of x.
    iterator insert_after(const_iterator position, size_type n, const T& x);
8
          Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).
9
          Effects: Inserts n copies of x after position.
10
          Returns: An iterator pointing to the last inserted copy of x or position if n == 0.
    template <class InputIterator>
      iterator insert_after(const_iterator position, InputIterator first, InputIterator last);
11
          Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).
         first and last are not iterators in *this.
12
          Effects: Inserts copies of elements in [first,last) after position.
13
          Returns: An iterator pointing to the last inserted element or position if first == last.
    iterator insert_after(const_iterator position, initializer_list<T> il);
14
          Effects: insert_after(p, il.begin(), il.end()).
15
          Returns: An iterator pointing to the last inserted element or position if i1 is empty.
    template <class... Args>
      iterator emplace_after(const_iterator position, Args&&... args);
16
          Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).
17
          Effects: Inserts an object of type value_type constructed with value_type(std::forward<Args>(
         args)...) after position.
18
    Returns: An iterator pointing to the new object.
    iterator erase_after(const_iterator position);
19
          Requires: The iterator following position is dereferenceable.
20
          Effects: Erases the element pointed to by the iterator following position.
21
          Returns: An iterator pointing to the element following the one that was erased, or end() if no such
         element exists.
22
          Throws: Nothing.
    iterator erase_after(const_iterator position, const_iterator last);
23
          Requires: All iterators in the range (position, last) are dereferenceable.
24
          Effects: Erases the elements in the range (position, last).
          Returns: last.
25
26
          Throws: Nothing.
                                                                                                          742
    § 23.3.4.5
```

```
void resize(size_type sz);
    void resize(size_type sz, const value_type& c);
27
         Effects: If sz < distance(begin(), end()), erases the last distance(begin(), end()) - sz ele-
         ments from the list. Otherwise, inserts sz - distance(begin(), end()) elements at the end of the
         list. For the first signature the inserted elements are value-initialized, and for the second signature
         they are copies of c.
28
         Requires: T shall be DefaultConstructible for the first form and it shall be CopyInsertable into
         *this for the second form.
    void clear() noexcept;
29
         Effects: Erases all elements in the range [begin(),end()).
30
         Remarks: Does not invalidate past-the-end iterators.
              forward_list operations
                                                                                           [forwardlist.ops]
    void splice_after(const_iterator position, forward_list& x);
    void splice_after(const_iterator position, forward_list&& x);
         Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(),end()).
         &x != this.
2
         Effects: Inserts the contents of x after position, and x becomes empty. Pointers and references to the
         moved elements of x now refer to those same elements but as members of *this. Iterators referring
         to the moved elements will continue to refer to their elements, but they now behave as iterators into
         *this, not into x.
3
         Throws: Nothing.
4
         Complexity: \mathcal{O}(distance(x.begin(), x.end()))
    void splice_after(const_iterator position, forward_list& x, const_iterator i);
    void splice_after(const_iterator position, forward_list&& x, const_iterator i);
5
         Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(),end()).
         The iterator following i is a dereferenceable iterator in x.
6
         Effects: Inserts the element following i into *this, following position, and removes it from x. The
         result is unchanged if position == i or position == ++i. Pointers and references to *i continue to
         refer to the same element but as a member of *this. Iterators to *i (including i itself) continue to
         refer to the same element, but now behave as iterators into *this, not into x.
7
         Throws: Nothing.
8
         Complexity: \mathcal{O}(1)
    void splice_after(const_iterator position, forward_list& x,
                       const_iterator first, const_iterator last);
    void splice_after(const_iterator position, forward_list&& x,
                       const_iterator first, const_iterator last);
```

§ 23.3.4.6 743

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.
- 11  $Complexity: \mathcal{O}(distance(first, last))$

```
void remove(const T& value);
template <class Predicate > void remove if(Predicate pred);
```

- Effects: Erases all the elements in the list referred by a list iterator i for which the following conditions hold: \*i == value (for remove()), pred(\*i) is true (for remove\_if()). 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. Invalidates only the iterators and references to the erased elements.
- 13 Throws: Nothing unless an exception is thrown by the equality comparison or the predicate.
- 14 Complexity: Exactly distance(begin(), end()) applications of the corresponding predicate.

```
void unique();
template <class BinaryPredicate> void unique(BinaryPredicate pred);
```

- Effects: Erases all but the first element from every consecutive group of equal elements referred to by the iterator i in the range [first + 1,last) for which \*i == \*(i-1) (for the version with no arguments) or pred(\*i, \*(i 1)) (for the version with a predicate argument) holds. Invalidates only the iterators and references to the erased elements.
- 16 Throws: Nothing unless an exception is thrown by the equality comparison or the predicate.
- Complexity: If the range [first,last) is not empty, exactly (last first) 1 applications of the corresponding predicate, otherwise no applications of the predicate.

```
void merge(forward_list& x);
void merge(forward_list& x);
template <class Compare> void merge(forward_list& x, Compare comp)
template <class Compare> void merge(forward_list& x, Compare comp)
```

- Requires: comp defines a strict weak ordering (25.4), and \*this and x are both sorted according to this ordering.
- Effects: Merges x into \*this. This operation shall be stable: for equivalent elements in the two lists, the elements from \*this shall always precede the elements from x. x is empty after the merge. If an exception is thrown other than by a comparison there are no effects. 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: At most distance(begin(), end()) + distance(x.begin(), x.end()) 1 comparisons.

```
void sort();
template <class Compare> void sort(Compare comp);
```

§ 23.3.4.6 744

21 Requires: operator< (for the version with no arguments) or comp (for the version with a comparison argument) defines a strict weak ordering (25.4).

- 22 Effects: Sorts the list according to the operator< or the comp function object. This operation shall be stable: the relative order of the equivalent elements is preserved. If an exception is thrown the order of the elements in \*this is unspecified. Does not affect the validity of iterators and references.
- Complexity: Approximately  $N \log N$  comparisons, where N is distance(begin(), end()).

```
void reverse() noexcept;
```

- *Effects:* Reverses the order of the elements in the list. Does not affect the validity of iterators and references.
- 25 Complexity: Linear time.

#### 23.3.4.7 forward list specialized algorithms

[forwardlist.spec]

```
template <class T, class Allocator>
  void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
      Effects: x.swap(y)
```

## 23.3.5 Class template list

[list]

## 23.3.5.1 Class template list overview

[list.overview]

- <sup>1</sup> A list is a sequence container that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Unlike vectors (23.3.6) and deques (23.3.3), fast random access to list elements is not supported, but many algorithms only need sequential access anyway.
- <sup>2</sup> A list satisfies all of the requirements of a container, of a reversible container (given in two tables in 23.2), of a sequence container, including most of the optional sequence container requirements (23.2.3), and of an allocator-aware container (Table 99). The exceptions are the operator[] and at member functions, which are not provided.<sup>264</sup> Descriptions are provided here only for operations on list that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <class T, class Allocator = allocator<T> >
  class list {
  public:
    // types:
    typedef value_type&
                                                                      reference;
    typedef const value_type&
                                                                      const_reference;
                                                                     // see 23.2
    typedef implementation-defined
                                                    iterator;
                                                    const_iterator; // see 23.2
    typedef implementation-defined
    typedef implementation-defined
                                                                    // see 23.2
                                                    size_type;
    typedef implementation-defined
                                                    difference_type; // see 23.2
    typedef T
                                                   value_type;
    typedef Allocator
                                                   allocator_type;
    typedef typename allocator_traits<Allocator>::pointer
                                                                      pointer;
    typedef typename allocator_traits<Allocator>::const_pointer
                                                                      const_pointer;
    typedef std::reverse_iterator<iterator>
                                                   reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

§ 23.3.5.1 745

<sup>264)</sup> These member functions are only provided by containers whose iterators are random access iterators.

```
// 23.3.5.2, 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& x);
list(list&& x);
list(const list&, const Allocator&);
list(list&&, const Allocator&);
list(initializer_list<T>, const Allocator& = Allocator());
~list();
list& operator=(const list& x);
list& operator=(list&& x);
list& operator=(initializer_list<T>);
template <class InputIterator>
  void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
void assign(initializer_list<T>);
allocator_type get_allocator() const noexcept;
// iterators:
iterator
                        begin() noexcept;
const_iterator
                        begin() const noexcept;
iterator
                        end() noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
                   rend() noexcept;
reverse_iterator
const_reverse_iterator rend() const noexcept;
const_iterator
                        cbegin() const noexcept;
const_iterator
                        cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;
// 23.3.5.3, capacity:
          empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
void
        resize(size_type sz);
void
          resize(size_type sz, const T& c);
// element access:
reference
                 front();
const_reference front() const;
reference
                back();
const_reference back() const;
// 23.3.5.4, modifiers:
template <class... Args> void emplace_front(Args&&... args);
void pop_front();
template <class... Args> void emplace_back(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
```

§ 23.3.5.1 746

```
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&);
  void
          clear() noexcept;
  // 23.3.5.5, list operations:
  void splice(const_iterator position, list& x);
  void splice(const_iterator position, list&& x);
  void splice(const_iterator position, list& x, const_iterator i);
  void splice(const_iterator position, list&& x, const_iterator i);
  void splice(const_iterator position, list& x,
              const_iterator first, const_iterator last);
  void splice(const_iterator position, list&& x,
              const_iterator first, const_iterator last);
  void remove(const T& value);
  template <class Predicate> void remove_if(Predicate pred);
  void unique();
  template <class BinaryPredicate>
    void unique(BinaryPredicate binary_pred);
 void merge(list& x);
  void merge(list&& x);
  template <class Compare> void merge(list& x, Compare comp);
  template <class Compare> void merge(list&& x, Compare comp);
 void sort();
 template <class Compare> void sort(Compare comp);
 void reverse() noexcept;
};
template <class T, class Allocator>
 bool operator==(const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class T, class Allocator>
 bool operator< (const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class T, class Allocator>
 bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class T, class Allocator>
  bool operator> (const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class T, class Allocator>
```

§ 23.3.5.1 747

```
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);
                                                                                                  [list.cons]
              list constructors, copy, and assignment
    explicit list(const Allocator& = Allocator());
1
         Effects: Constructs an empty list, using the specified allocator.
2
         Complexity: Constant.
    explicit list(size_type n);
3
         Effects: Constructs a list with n value-initialized elements.
4
         Requires: T shall be DefaultConstructible.
5
         Complexity: Linear in n.
    list(size_type n, const T& value,
         const Allocator& = Allocator());
6
         Effects: Constructs a list with n copies of value, using the specified allocator.
7
         Requires: T shall be CopyInsertable into *this.
8
         Complexity: Linear in n.
    template <class InputIterator>
    list(InputIterator first, InputIterator last,
         const Allocator& = Allocator());
9
         Effects: Constructs a list equal to the range [first,last).
10
         Complexity: Linear in distance(first, last).
    template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
11
         Effects: Replaces the contents of the list with the range [first, last).
    void assign(size_type n, const T& t);
12
         Effects: Replaces the contents of the list with n copies of t.
```

§ 23.3.5.2 748

```
23.3.5.3 list capacity
                                                                                            [list.capacity]
   void resize(size_type sz);
1
        Effects: If size() < sz, appends sz - size() value-initialized elements to the sequence. If sz <=
        size(), equivalent to
          list<T>::iterator it = begin();
          advance(it, sz);
          erase(it, end());
2
        Requires: T shall be DefaultConstructible.
   void resize(size_type sz, const T& c);
3
        Effects:
          if (sz > size())
            insert(end(), sz-size(), c);
          else if (sz < size()) {
            iterator i = begin();
            advance(i, sz);
             erase(i, end());
          }
          else
                               // do nothing
             ;
        Requires: T shall be CopyInsertable into *this.
                                                                                           [list.modifiers]
   23.3.5.4 list modifiers
   iterator insert(const_iterator position, const T& x);
   iterator insert(const_iterator position, T&& x);
   iterator insert(const_iterator position, size_type n, const T& x);
   template <class InputIterator>
     iterator insert(const_iterator position, InputIterator first,
                      InputIterator last);
   iterator insert(const_iterator position, initializer_list<T>);
   template <class... Args> void emplace_front(Args&&... args);
   template <class... Args> void emplace_back(Args&&... args);
   template <class... Args> iterator emplace(const_iterator position, Args&&... args);
   void push_front(const T& x);
   void push_front(T&& x);
   void push_back(const T& x);
   void push_back(T&& x);
1
        Remarks: Does not affect the validity of iterators and references. If an exception is thrown there are
        no effects.
2
         Complexity: Insertion of a single element into a list takes constant time and exactly one call to a
        constructor of T. Insertion of multiple elements into a list is linear in the number of elements inserted,
        and the number of calls to the copy constructor or move constructor of T is exactly equal to the number
```

§ 23.3.5.4 749

of elements inserted.

```
iterator erase(const_iterator position);
   iterator erase(const_iterator first, const_iterator last);
   void pop_front();
   void pop_back();
   void clear() noexcept;
3
         Effects: Invalidates only the iterators and references to the erased elements.
         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.
   23.3.5.5 list operations
                                                                                                    [list.ops]
  Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically
   for them.<sup>265</sup>
  list provides three splice operations that destructively move elements from one list to another. The behavior
   of splice operations is undefined if get_allocator() != x.get_allocator().
   void splice(const_iterator position, list& x);
   void splice(const_iterator position, list&& x);
3
         Requires: &x != this.
         Effects: Inserts the contents of x before position and x becomes empty. Pointers and references to
        the moved elements of x now refer to those same elements but as members of *this. Iterators referring
        to the moved elements will continue to refer to their elements, but they now behave as iterators into
         *this, not into x.
         Throws: Nothing.
         Complexity: Constant time.
   void splice(const_iterator position, list& x, const_iterator i);
   void splice(const_iterator position, list&& x, const_iterator i);
        Effects: Inserts an element pointed to by i from list x before position and removes the element from
```

7 x. The result is unchanged if position == i or position == ++i. Pointers and references to \*i continue to refer to this same element but as a member of \*this. Iterators to \*i (including i itself) continue to refer to the same element, but now behave as iterators into \*this, not into x.

- 8 Requires: i is a valid dereferenceable iterator of x.
- 9 Throws: Nothing.

4

5

4

5

6

10 Complexity: Constant time.

```
void splice(const_iterator position, list& x, const_iterator first,
            const_iterator last);
void splice(const_iterator position, list&& x, const_iterator first,
            const_iterator last);
```

265) As specified in 17.6.3.5, the requirements in this Clause apply only to lists whose allocators compare equal.

§ 23.3.5.5 750

Effects: Inserts elements in the range [first,last) before position and removes the elements from x.

- Requires: [first, last) is a valid range in x. The result is undefined if position is an iterator in the range [first,last). Pointers and references to the moved elements of x now refer to those same elements but as members of \*this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into \*this, not into x.
- 13 Throws: Nothing.
- 14 Complexity: Constant time if &x == this; otherwise, linear time.

```
void remove(const T& value);
template <class Predicate> void remove_if(Predicate pred);
```

- Effects: Erases all the elements in the list referred by a list iterator i for which the following conditions hold: \*i == value, pred(\*i) != false. Invalidates only the iterators and references to the erased elements.
- 16 Throws: Nothing unless an exception is thrown by \*i == value or pred(\*i) != false.
- 17 Remarks: Stable.
- 18 Complexity: Exactly size() applications of the corresponding predicate.

```
void unique();
template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

- Effects: Erases all but the first element from every consecutive group of equal elements referred to by the iterator i in the range [first + 1,last) for which \*i == \*(i-1) (for the version of unique with no arguments) or pred(\*i, \*(i 1)) (for the version of unique with a predicate argument) holds. Invalidates only the iterators and references to the erased elements.
- 20 Throws: Nothing unless an exception in thrown by \*i == \*(i-1) or pred(\*i, \*(i 1))
- 21 Complexity: If the range [first, last) is not empty, exactly (last first) 1 applications of the corresponding predicate, otherwise no applications of the predicate.

- Requires: comp shall define a strict weak ordering (25.4), and both the list and the argument list shall be sorted according to this ordering.
- Effects: If (&x == this) does nothing; otherwise, merges the two sorted ranges [begin(), end()) and [x.begin(), x.end()). The result is a range in which the elements will be sorted in non-decreasing order according to the ordering defined by comp; that is, for every iterator i, in the range other than the first, the condition comp(\*i, \*(i 1)) will be false. Pointers and references to the moved elements of x now refer to those same elements but as members of \*this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into \*this, not into x.
- Remarks: Stable. If (&x != this) the range [x.begin(), x.end()) is empty after the merge. No elements are copied by this operation. The behavior is undefined if this->get\_allocator() != x.get\_allocator().

§ 23.3.5.5 751

Complexity: At most size() + x.size() - 1 applications of comp if (&x != this); otherwise, no applications of comp are performed. If an exception is thrown other than by a comparison there are no effects.

```
void reverse() noexcept;
```

- 26 Effects: Reverses the order of the elements in the list. Does not affect the validity of iterators and references.
- 27 Complexity: Linear time.

```
void sort();
template <class Compare> void sort(Compare comp);
```

- Requires: operator< (for the first version) or comp (for the second version) shall define a strict weak ordering (25.4).
- 29 Effects: Sorts the list according to the operator< or a Compare function object. Does not affect the validity of iterators and references.
- 30 Remarks: Stable.
- Complexity: Approximately  $N \log(N)$  comparisons, where N == size().

## 23.3.5.6 list specialized algorithms

[list.special]

```
template <class T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>& y);

Effects:
    x.swap(y);
```

# 23.3.6 Class template vector

[vector]

# 23.3.6.1 Class template vector overview

[vector.overview]

- 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 99). The exceptions are the push\_front, pop\_front, and emplace\_front member functions, which are not provided. Descriptions are provided here only for operations on vector that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <class T, class Allocator = allocator<T> >
  class vector {
  public:
    // types:
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef implementation-defined iterator; // see 23.2
```

§ 23.3.6.1 752

```
const_iterator; // see 23.2
typedef implementation-defined
                                                              // see 23.2
\verb|typedef| implementation-defined|
                                             size_type;
                                              difference_type; // see 23.2
typedef implementation-defined
typedef T
                                               value_type;
typedef Allocator
                                               allocator_type;
typedef typename allocator_traits<Allocator>::pointer
                                                                 pointer;
typedef typename allocator_traits<Allocator>::const_pointer
                                                                 const_pointer;
typedef std::reverse iterator<iterator>
                                             reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
// 23.3.6.2, construct/copy/destroy:
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& x);
vector(vector&&);
vector(const vector&, const Allocator&);
vector(vector&&, const Allocator&);
vector(initializer_list<T>, const Allocator& = Allocator());
~vector():
vector& operator=(const vector& x);
vector& operator=(vector&& x);
vector& operator=(initializer_list<T>);
template <class InputIterator>
  void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& u);
void assign(initializer_list<T>);
allocator_type get_allocator() const noexcept;
// iterators:
iterator
                       begin() noexcept;
const_iterator
                     begin() const noexcept;
                      end() noexcept;
iterator
                      end() const noexcept;
const_iterator
reverse_iterator
                       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
                       cbegin() const noexcept;
const_iterator
const_iterator
                       cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;
// 23.3.6.3, capacity:
size_type size() const noexcept;
size_type max_size() const noexcept;
void
          resize(size_type sz);
void
          resize(size_type sz, const T& c);
size_type capacity() const noexcept;
        empty() const noexcept;
          reserve(size_type n);
void
```

§ 23.3.6.1 753

```
void
                shrink_to_fit();
      // element access:
                      operator[](size_type n);
      reference
      const_reference operator[](size_type n) const;
      const_reference at(size_type n) const;
     reference
                     at(size_type n);
     reference
                     front();
     const_reference front() const;
     reference
                    back();
      const_reference back() const;
      // 23.3.6.4, data access
                data() noexcept;
     const T* data() const noexcept;
      // 23.3.6.5, modifiers:
      template <class... Args> void emplace_back(Args&&... args);
      void push_back(const T& x);
      void push_back(T&& x);
      void pop_back();
      template <class... Args> iterator emplace(const_iterator position, Args&&... args);
      iterator insert(const_iterator position, const T& x);
      iterator
                   insert(const_iterator position, T&& x);
                   insert(const_iterator position, size_type n, const T& x);
      template <class InputIterator>
          iterator insert(const_iterator position,
                          InputIterator first, InputIterator last);
                   insert(const_iterator position, initializer_list<T> il);
     iterator erase(const_iterator position);
      iterator erase(const_iterator first, const_iterator last);
               swap(vector&);
      void
               clear() noexcept;
    };
    template <class T, class Allocator>
     bool operator==(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
    template <class T, class Allocator>
     bool operator< (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
    template <class T, class Allocator>
     bool operator!=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
    template <class T, class Allocator>
     bool operator> (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
    template <class T, class Allocator>
     bool operator>=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
    template <class T, class Allocator>
     bool operator<=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
    // 23.3.6.6, specialized algorithms:
    template <class T, class Allocator>
     void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
23.3.6.2 vector constructors, copy, and assignment
                                                                                       [vector.cons]
```

§ 23.3.6.2 754

```
explicit vector(const Allocator& = Allocator());
         Effects: Constructs an empty vector, using the specified allocator.
2
          Complexity: Constant.
    explicit vector(size_type n);
3
         Effects: Constructs a vector with n value-initialized elements.
4
          Requires: T shall be DefaultConstructible.
5
          Complexity: Linear in n.
    vector(size_type n, const T& value,
           const Allocator& = Allocator());
6
         Effects: Constructs a vector with n copies of value, using the specified allocator.
7
          Requires: T shall be CopyInsertable into *this.
8
          Complexity: Linear in n.
    template <class InputIterator>
      vector(InputIterator first, InputIterator last,
             const Allocator& = Allocator());
9
         Effects: Constructs a vector equal to the range [first,last), using the specified allocator.
10
          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);
11
         Effects:
           erase(begin(), end());
           insert(begin(), first, last);
    void assign(size_type n, const T& t);
12
          Effects:
           erase(begin(), end());
           insert(begin(), n, t);
```

§ 23.3.6.2 755

```
23.3.6.3 vector capacity
```

[vector.capacity]

size\_type capacity() const noexcept;

Returns: The total number of elements that the vector can hold without requiring reallocation.

```
void reserve(size_type n);
```

1

- 2 Effects: A directive that informs a vector of a planned change in size, so that it can manage the storage allocation accordingly. After reserve(), capacity() is greater or equal to the argument of reserve if reallocation happens; and equal to the previous value of capacity() otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of reserve(). If an exception is thrown other than by the move constructor of a non-CopyInsertable type, there are no effects.
- 3 Complexity: It does not change the size of the sequence and takes at most linear time in the size of the sequence.
- 4 Throws: length\_error if n > max\_size(). 266
- 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& x);
```

- 7 Effects: Exchanges the contents and capacity() of \*this with that of x.
- 8 Complexity: Constant time.

```
void resize(size_type sz);
```

- Effects: If sz <= size(), equivalent to erase(begin() + sz, end());. If size() < sz, appends sz size() value-initialized elements to the sequence.</p>
- 10 Requires: T shall be CopyInsertable into \*this.

```
void resize(size_type sz, const T& c);
```

11 Effects:

Requires: If an exception is thrown other than by the move constructor of a non-CopyInsertable T there are no effects.

§ 23.3.6.3 756

<sup>266)</sup> reserve() uses Allocator::allocate() which may throw an appropriate exception.

```
[vector.data]
   23.3.6.4
            vector data
   Т*
              data() noexcept;
   const T*
              data() const noexcept;
1
        Returns: A pointer such that [data(),data() + size()) is a valid range. For a non-empty vector,
        data() == &front().
2
        Complexity: Constant time.
   23.3.6.5
             vector modifiers
                                                                                       [vector.modifiers]
   iterator insert(const_iterator position, const T& x);
   iterator insert(const_iterator position, T&& x);
   iterator insert(const_iterator position, size_type n, const T& x);
   template <class InputIterator>
     iterator insert(const_iterator position, InputIterator first, InputIterator last);
   iterator insert(const_iterator position, initializer_list<T>);
   template <class... Args> void emplace_back(Args&&... args);
   template <class... Args> iterator emplace(const_iterator position, Args&&... args);
   void push_back(const T& x);
   void push_back(T&& x);
1
        Remarks: Causes reallocation if the new size is greater than the old capacity. If no reallocation happens,
        all the iterators and references before the insertion point remain valid. If an exception is thrown other
        than by the copy constructor, move constructor, assignment operator, or move assignment operator
        of T or by any InputIterator operation there are no effects. If an exception is thrown by the move
        constructor of a non-CopyInsertable T, the effects are unspecified.
2
        Complexity: The complexity is linear in the number of elements inserted plus the distance to the end
        of the vector.
   iterator erase(const_iterator position);
```

- Effects: Invalidates iterators and references at or after the point of the erase.
- <sup>4</sup> 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.

## 23.3.6.6 vector specialized algorithms

[vector.special]

```
template <class T, class Allocator>
    void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);

Effects:
    x.swap(y);
```

iterator erase(const\_iterator first, const\_iterator last);

#### 23.3.7 Class vector<bool>

3

[vector.bool]

<sup>1</sup> To optimize space allocation, a specialization of vector for bool elements is provided:

§ 23.3.7 757

```
namespace std {
  template <class Allocator> class vector<bool, Allocator> {
 public:
    // types:
    typedef bool
                                                   const_reference;
    typedef implementation-defined
                                                                   // see 23.2
                                                   iterator;
    typedef implementation-defined
                                                   const_iterator; // see 23.2
                                                  size_type; // see 23.2
    typedef implementation-defined
    typedef implementation-defined
                                                   difference_type; // see 23.2
    typedef bool
                                                   value_type;
    typedef Allocator
                                                   allocator_type;
                                                   pointer;
    typedef implementation-defined
    typedef implementation-defined
                                                   const_pointer;
    typedef std::reverse_iterator<iterator>
                                                   reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    // bit reference:
    class reference {
      friend class vector;
      reference() noexcept;
    public:
      ~reference();
      operator bool() const noexcept;
      reference& operator=(const bool x) noexcept;
      reference& operator=(const reference& x) noexcept;
      void flip() noexcept;
                               // flips the bit
    };
    // construct/copy/destroy:
    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);
    vector<bool,Allocator>& operator=(vector<bool,Allocator>&& x);
    vector operator=(initializer_list<bool>);
    template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const bool& t);
    void assign(initializer_list<bool>;
    allocator_type get_allocator() const noexcept;
    // iterators:
    iterator
                           begin() noexcept;
    const_iterator
                           begin() const noexcept;
    iterator
                           end() noexcept;
                           end() const noexcept;
    const_iterator
```

§ 23.3.7 758

```
reverse_iterator
                       rbegin() noexcept;
 const_reverse_iterator rbegin() const noexcept;
 reverse_iterator
                       rend() noexcept;
  const_reverse_iterator rend() const noexcept;
  const_iterator
                         cbegin() const noexcept;
  const_iterator
                         cend() const noexcept;
  const reverse iterator crbegin() const noexcept;
  const_reverse_iterator crend() const noexcept;
  // capacity:
  size_type size() const noexcept;
  size_type max_size() const noexcept;
           resize(size_type sz, bool c = false);
  size_type capacity() const noexcept;
            empty() const noexcept;
  bool
  void
            reserve(size_type n);
  void
            shrink_to_fit();
  // element access:
                  operator[](size_type n);
  reference
  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) noexcept;
 void flip() noexcept;
                             // flips all bits
  void clear() noexcept;
};
```

- Unless described below, all operations have the same requirements and semantics as the primary vector template, except that operations dealing with the bool value type map to bit values in the container storage and allocator\_traits::construct (20.6.8.2) is not used to construct these values.
- <sup>3</sup> 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.
- <sup>4</sup> 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

§ 23.3.7

```
the argument is (convertible to) true and clears it otherwise. flip reverses the state of the bit.
   void flip() noexcept;
        Effects: Replaces each element in the container with its complement.
   static void swap(reference x, reference y) noexcept;
6
        Effects: exchanges the contents of x and y as if by
          bool b = x;
          x = y;
          y = b;
   template <class Allocator> struct hash<vector<bool, Allocator> >;
        Requires: the template specialization shall meet the requirements of class template hash (20.8.12).
   23.4 Associative containers
                                                                                           [associative]
   23.4.1
           In general
                                                                                  [associative.general]
  The header <map> defines the class templates map and multimap; the header <set> defines the class templates
   set and multiset.
   23.4.2
            Header <map> synopsis
```

[associative.map.syn]

```
#include <initializer_list>
namespace std {
  template <class Key, class T, class Compare = less<Key>,
            class Allocator = allocator<pair<const Key, T> > >
    class map;
  template <class Key, class T, class Compare, class Allocator>
    bool operator == (const map < Key, T, Compare, Allocator > & x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator> (const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
  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>,
```

§ 23.4.2 760

```
class Allocator = allocator<pair<const Key, T> > >
      class multimap;
   template <class Key, class T, class Compare, class Allocator>
     bool operator == (const multimap < Key, T, Compare, Allocator > & x,
                      const multimap<Key,T,Compare,Allocator>& y);
   template <class Key, class T, class Compare, class Allocator>
     bool operator< (const multimap<Key,T,Compare,Allocator>& x,
                      const multimap<Key,T,Compare,Allocator>& y);
   template <class Key, class T, class Compare, class Allocator>
      bool operator!=(const multimap<Key,T,Compare,Allocator>& x,
                      const multimap<Key,T,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
      bool operator> (const multimap<Key,T,Compare,Allocator>& x,
                      const multimap<Key,T,Compare,Allocator>& y);
   template <class Key, class T, class Compare, class Allocator>
      bool operator>=(const multimap<Key,T,Compare,Allocator>& x,
                      const multimap<Key,T,Compare,Allocator>& y);
   template <class Key, class T, class Compare, class Allocator>
     bool operator <= (const multimap < Key, T, Compare, Allocator > & x,
                      const multimap<Key,T,Compare,Allocator>& y);
   template <class Key, class T, class Compare, class Allocator>
      void swap(multimap<Key,T,Compare,Allocator>& x,
                multimap<Key,T,Compare,Allocator>& y);
 }
23.4.3
                                                                              [associative.set.syn]
        Header <set> synopsis
 #include <initializer_list>
 namespace std {
   template <class Key, class Compare = less<Key>,
              class Allocator = allocator<Key> >
      class set:
   template <class Key, class Compare, class Allocator>
      bool operator==(const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
   template <class Key, class Compare, class Allocator>
      bool operator< (const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
   template <class Key, class Compare, class Allocator>
      bool operator!=(const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
   template <class Key, class Compare, class Allocator>
     bool operator> (const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
   template <class Key, class Compare, class Allocator>
     bool operator>=(const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
   template <class Key, class Compare, class Allocator>
      bool operator<=(const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
    template <class Key, class Compare, class Allocator>
      void swap(set<Key,Compare,Allocator>& x,
                set<Key,Compare,Allocator>& y);
```

§ 23.4.3 761

```
template <class Key, class Compare = less<Key>,
            class Allocator = allocator<Key> >
    class multiset;
  template <class Key, class Compare, class Allocator>
    bool operator == (const multiset < Key, Compare, Allocator > & x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
 template <class Key, class Compare, class Allocator>
    bool operator!=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator> (const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator>=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator <= (const multiset < Key, Compare, Allocator > & x,
                    const multiset<Key,Compare,Allocator>& y);
 template <class Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator>& x,
              multiset<Key,Compare,Allocator>& y);
}
```

## 23.4.4 Class template map

[map]

# 23.4.4.1 Class template map overview

[map.overview]

- A map is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type T based on the keys. The map class supports bidirectional iterators
- A map satisfies all of the requirements of a container, of a reversible container (23.2.4), of an associative container (23.2.4), and of an allocator-aware container (Table 99). A map also provides most operations described in (23.2.4) for unique keys. This means that a map supports the a\_uniq operations in (23.2.4) but not the a\_eq operations. For a map<Key,T> the key\_type is Key and the value\_type is pair<const Key,T>. Descriptions are provided here only for operations on map that are not described in one of those tables or for operations where there is additional semantic information.

```
namespace std {
  template <class Key, class T, class Compare = less<Key>,
             class Allocator = allocator<pair<const Key, T> > >
  class map {
  public:
    // types:
    typedef Key
                                                      key_type;
                                                      mapped_type;
    typedef T
    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;
                                                                        // see 23.2
    typedef implementation-defined
                                                       iterator;
    {\tt typedef} \ \textit{implementation-defined}
                                                       const_iterator; // see 23.2
                                                                        // see 23.2
    {\tt typedef} \ \textit{implementation-defined}
                                                       size_type;
    typedef implementation-defined
                                                       difference_type; // see 23.2
```

§ 23.4.4.1 762

```
typedef typename allocator_traits<Allocator>::pointer
                                                                  pointer;
typedef typename allocator_traits<Allocator>::const_pointer
                                                                  const_pointer;
typedef std::reverse_iterator<iterator>
                                              reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
class value_compare {
friend class map;
protected:
  Compare comp;
  value_compare(Compare c) : comp(c) {}
public:
  typedef bool result_type;
  typedef value_type first_argument_type;
  typedef value_type second_argument_type;
  bool operator()(const value_type& x, const value_type& y) const {
    return comp(x.first, y.first);
  }
};
// 23.4.4.2, construct/copy/destroy:
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);
explicit 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 noexcept;
// iterators:
iterator
                        begin() noexcept;
const_iterator
                        begin() const noexcept;
iterator
                        end() noexcept;
const_iterator
                       end() const noexcept;
reverse_iterator
                       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
                        cbegin() const noexcept;
const_iterator
const_iterator
                        cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
```

§ 23.4.4.1 763

```
const_reverse_iterator crend() const noexcept;
  // capacity:
            empty() const noexcept;
  bool
  size_type size() const noexcept;
  size_type max_size() const noexcept;
  // 23.4.4.3, element access:
  T& operator[](const key_type& x);
 T& operator[](key_type&& x);
           at(const key_type& x);
  const T& at(const key_type& x) const;
  // 23.4.4.4, modifiers:
  template <class... Args> pair<iterator, bool> emplace(Args&&... args);
  template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
  pair<iterator, bool> insert(const value_type& x);
  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() noexcept;
  // observers:
  key_compare
               key_comp() const;
  value_compare value_comp() const;
  // 23.4.4.5, map operations:
  iterator
            find(const key_type& x);
  const_iterator find(const key_type& x) const;
                count(const key_type& x) const;
  size_type
  iterator
                lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
                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>
```

§ 23.4.4.1 764

```
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);
     }
   23.4.4.2
            map constructors, copy, and assignment
                                                                                              [map.cons]
   explicit map(const Compare& comp = Compare(),
                const Allocator& = Allocator());
1
        Effects: Constructs an empty map using the specified comparison object and allocator.
2
        Complexity: Constant.
   template <class InputIterator>
     map(InputIterator first, InputIterator last,
         const Compare& comp = Compare(), const Allocator& = Allocator());
3
        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.
4
        Effects: Constructs an empty map 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.
   23.4.4.3 map element access
                                                                                            [map.access]
   T& operator[](const key_type& x);
1
        Effects: If there is no key equivalent to x in the map, inserts value_type(x, T()) into the map.
2
        Requires: key_type shall be CopyConstructible and mapped_type shall be DefaultConstructible.
3
        Returns: A reference to the mapped_type corresponding to x in *this.
        Complexity: logarithmic.
   T& operator[](key_type&& x);
```

§ 23.4.4.3 765

```
5
         Effects: If there is no key equivalent to x in the map, inserts value_type(std::move(x), T()) into
         the map.
6
         Requires: mapped_type shall be DefaultConstructible.
         Returns: A reference to the mapped_type corresponding to x in *this.
         Complexity: logarithmic.
             at(const key_type& x);
   const T& at(const key_type& x) const;
9
         Returns: A reference to the mapped_type corresponding to x in *this.
10
         Throws: An exception object of type out_of_range if no such element is present.
11
         Complexity: logarithmic.
   23.4.4.4 map modifiers
                                                                                         [map.modifiers]
   template <class P> pair<iterator, bool> insert(P&& x);
   template <class P> pair<iterator, bool> insert(const_iterator position, P&& x);
   template <class InputIterator>
     void insert(InputIterator first, InputIterator last);
```

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<key\_type,mapped\_type>. Otherwise CopyConstructible is required for both key\_type and mapped\_type.

## 23.4.4.5 map operations

1

[map.ops]

```
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_iterator upper_bound(const key_type & x) const;

pair<iterator, iterator>
    equal_range(const key_type & x);
pair<const_iterator, const_iterator>
    equal_range(const key_type& x) const;

The find lower bound upper bound and equal
```

Requires: P shall be convertible to value\_type.

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).

§ 23.4.4.5

#### 23.4.4.6 map specialized algorithms

[map.special]

## 23.4.5 Class template multimap

[multimap]

# 23.4.5.1 Class template multimap overview

[multimap.overview]

- A multimap is an associative container that supports equivalent keys (possibly containing multiple copies of the same key value) and provides for fast retrieval of values of another type T based on the keys. The multimap class supports bidirectional iterators.
- A multimap satisfies all of the requirements of a container and of a reversible container (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 99). A multimap also provides most operations described in (23.2.4) for equal keys. This means that a multimap supports the a\_eq operations 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
                                                                   // see 23.2
                                                   iterator;
    typedef implementation-defined
                                                   const_iterator; // see 23.2
    typedef implementation-defined
                                                                   // see 23.2
                                                   size_type;
    typedef implementation-defined
                                                   difference_type; // see 23.2
    typedef typename allocator_traits<Allocator>::pointer
                                                                     pointer;
    typedef typename allocator_traits<Allocator>::const_pointer
                                                                     const_pointer;
    typedef std::reverse_iterator<iterator>
                                                  reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    class value_compare {
    friend class multimap;
    protected:
      Compare comp;
      value_compare(Compare c) : comp(c) { }
      typedef bool result_type;
      typedef value_type first_argument_type;
      typedef value_type second_argument_type;
      bool operator()(const value_type& x, const value_type& y) const {
        return comp(x.first, y.first);
```

§ 23.4.5.1 767

```
}
}:
// construct/copy/destroy:
explicit multimap(const Compare& comp = Compare(),
                   const Allocator& = Allocator());
template <class InputIterator>
  multimap(InputIterator first, InputIterator last,
           const Compare& comp = Compare(),
           const Allocator& = Allocator());
multimap(const multimap<Key,T,Compare,Allocator>& x);
multimap(multimap<Key,T,Compare,Allocator>&& x);
explicit multimap(const Allocator&);
multimap(const multimap&, const Allocator&);
multimap(multimap&&, const Allocator&);
multimap(initializer_list<value_type>,
  const Compare& = Compare(),
  const Allocator& = Allocator());
~multimap();
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>);
allocator_type get_allocator() const noexcept;
// iterators:
iterator
                        begin() noexcept;
                       begin() const noexcept;
const_iterator
                        end() noexcept;
iterator
const_iterator
                      end() const noexcept;
reverse_iterator
                      rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator
                      rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_iterator
                        cbegin() const noexcept;
const_iterator
                        cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;
// capacity:
               empty() const noexcept;
bool
size_type
               size() const noexcept;
size_type
               max_size() const noexcept;
// modifiers:
template <class... Args> iterator emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
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>
```

§ 23.4.5.1 768

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() noexcept;
      // observers:
      key_compare
                     key_comp() const;
      value_compare value_comp() const;
      // map operations:
                    find(const key_type& x);
      iterator
      const_iterator find(const key_type& x) const;
                   count(const key_type& x) const;
      size_type
                     lower_bound(const key_type& x);
      iterator
      const_iterator lower_bound(const key_type& x) const;
                     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:
    template <class Key, class T, class Compare, class Allocator>
      void swap(multimap<Key,T,Compare,Allocator>& x,
                multimap<Key,T,Compare,Allocator>& y);
23.4.5.2 multimap constructors
                                                                                    [multimap.cons]
§ 23.4.5.2
                                                                                                  769
```

```
explicit multimap(const Compare& comp = Compare(),
                     const Allocator& = Allocator());
1
        Effects: Constructs an empty multimap using the specified comparison object and allocator.
2
        Complexity: Constant.
   template <class InputIterator>
     multimap(InputIterator first, InputIterator last,
              const Compare& comp = Compare(),
              const Allocator& = Allocator());
3
        Requires: If the iterator's 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).
5
        Complexity: Linear in N if the range [first,last) is already sorted using comp and otherwise N \log N,
        where N is last - first.
   23.4.5.3 multimap modifiers
                                                                                  [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 con-
        sidered 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<key_type, mapped_type>. Otherwise CopyConstructible is required for both
        key_type and mapped_type.
```

## 23.4.5.4 multimap operations

[multimap.ops]

```
iterator
               find(const key_type &x);
const_iterator find(const key_type& x) const;
iterator
               lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
pair<iterator, iterator>
  equal_range(const key_type& x);
pair<const_iterator, const_iterator>
  equal_range(const key_type& x) const;
```

The find, lower\_bound, upper\_bound, and equal\_range member functions each have two versions, one const and one non-const. In each case the behavior of the two versions is identical except that the const version returns a const\_iterator and the non-const version an iterator (23.2.4).

#### 23.4.5.5multimap specialized algorithms

[multimap.special]

```
template <class Key, class T, class Compare, class Allocator>
  void swap(multimap<Key,T,Compare,Allocator>& x,
           multimap<Key,T,Compare,Allocator>& y);
```

§ 23.4.5.5 770

## 23.4.6 Class template set

[set]

## 23.4.6.1 Class template set overview

[set.overview]

A set is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves. The set class supports bidirectional iterators.

A set satisfies all of the requirements of a container, of a reversible container (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 99). A set also provides most operations described in (23.2.4) for unique keys. This means that a set supports the a\_uniq operations in (23.2.4) but not the a\_eq operations. For a set<Key> both the key\_type and value\_type are Key. Descriptions are provided here only for operations on set that are not described in one of these tables and for operations where there is additional semantic information.

```
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
    {\tt typedef} \ \textit{implementation-defined}
                                                    const_iterator; // See 23.2
                                                                    // See 23.2
    typedef implementation-defined
                                                    size_type;
    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;
    // 23.4.6.2, 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);
    explicit 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);
```

§ 23.4.6.1

```
set<Key,Compare,Allocator>& operator=
  (set<Key,Compare,Allocator>&& x);
set& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;
// iterators:
iterator
                       begin() noexcept;
const iterator
                       begin() const noexcept;
                       end() noexcept;
iterator
                       end() const noexcept;
const_iterator
reverse_iterator
                       rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
                     rend() noexcept;
reverse_iterator
const_reverse_iterator rend() const noexcept;
const_iterator
                       cbegin() const noexcept;
const_iterator
                       cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;
// capacity:
              empty() const noexcept;
bool
             size() const noexcept;
size_type
size_type
             max_size() const noexcept;
// modifiers:
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator,bool> insert(const value_type& x);
pair<iterator,bool> insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(set<Key,Compare,Allocator>&);
void clear() noexcept;
// observers:
key_compare
             key_comp() const;
value_compare value_comp() const;
// set operations:
                find(const key_type& x);
iterator
const_iterator find(const key_type& x) const;
size_type count(const key_type& x) const;
                lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
```

§ 23.4.6.1 772

```
upper_bound(const key_type& x);
         iterator
         const_iterator upper_bound(const key_type& x) const;
         pair<iterator,iterator>
                                              equal_range(const key_type& x);
         pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
       };
       template <class Key, class Compare, class Allocator>
         bool operator==(const set<Key,Compare,Allocator>& x,
                         const set<Key,Compare,Allocator>& y);
       template <class Key, class Compare, class Allocator>
         bool operator< (const set<Key,Compare,Allocator>& x,
                         const set<Key,Compare,Allocator>& y);
       template <class Key, class Compare, class Allocator>
         bool operator!=(const set<Key,Compare,Allocator>& x,
                         const set<Key,Compare,Allocator>& y);
       template <class Key, class Compare, class Allocator>
         bool operator> (const set<Key,Compare,Allocator>& x,
                         const set<Key,Compare,Allocator>& y);
       template <class Key, class Compare, class Allocator>
         bool operator>=(const set<Key,Compare,Allocator>& x,
                         const set<Key,Compare,Allocator>& y);
       template <class Key, class Compare, class Allocator>
         bool operator<=(const set<Key,Compare,Allocator>& x,
                         const set<Key,Compare,Allocator>& y);
       // specialized algorithms:
       template <class Key, class Compare, class Allocator>
         void swap(set<Key,Compare,Allocator>& x,
                   set<Key,Compare,Allocator>& y);
    }
                                                                                               [set.cons]
   23.4.6.2
            set constructors, copy, and assignment
   explicit set(const Compare& comp = Compare(),
                const Allocator& = Allocator());
1
        Effects: Constructs an empty set using the specified comparison objects and allocator.
2
        Complexity: Constant.
   template <class InputIterator>
     set(InputIterator first, InputIterator last,
         const Compare& comp = Compare(), const Allocator& = Allocator());
3
        Effects: Constructs an empty set using the specified comparison object and allocator, and inserts
        elements from the range [first,last).
        Requires: If the iterator's dereference operator returns an Ivalue or a non-const rvalue, then Key shall
        be CopyConstructible.
5
        Complexity: Linear in N if the range [first,last) is already sorted using comp and otherwise N \log N,
        where N is last - first.
   23.4.6.3
             set specialized algorithms
                                                                                            [set.special]
```

§ 23.4.6.3

# 23.4.7 Class template multiset

[multiset]

## 23.4.7.1 Class template multiset overview

[multiset.overview]

- A multiset is an associative container that supports equivalent keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves. The multiset class supports bidirectional iterators.
- A multiset satisfies all of the requirements of a container, of a reversible container (23.2), of an associative container (23.2.4), and of an allocator-aware container (Table 99). multiset also provides most operations described in (23.2.4) for duplicate keys. This means that a multiset supports the a\_eq operations in (23.2.4) but not the a\_uniq operations. For a multiset<Key> both the key\_type and value\_type are Key. Descriptions are provided here only for operations on multiset that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <class Key, class Compare = less<Key>,
             class Allocator = allocator<Key> >
  class multiset {
  public:
    // types:
    typedef Key
                                                                         key_type;
    typedef Key
                                                                         value_type;
    typedef Compare
                                                                         key_compare;
    typedef Compare
                                                                         value_compare;
    typedef Allocator
                                                                         allocator_type;
    typedef value_type&
                                                                         reference;
    typedef const value_type&
                                                                         const_reference;
    typedef implementation-defined
                                                     iterator;
                                                                       // see 23.2
    {\tt typedef} \ \textit{implementation-defined}
                                                      const_iterator; // see 23.2
                                                                       // see 23.2
    {\tt typedef} \ \textit{implementation-defined}
                                                      size_type;
                                                      {\tt difference\_type;} /\!/ \, see \, {\tt 23.2}
    \verb"typedef" implementation-defined"
    typedef typename allocator_traits<Allocator>::pointer
                                                                         pointer;
    typedef typename allocator_traits<Allocator>::const_pointer
                                                                         const_pointer;
    typedef std::reverse_iterator<iterator>
                                                     reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    // construct/copy/destroy:
    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);
    explicit multiset(const Allocator&);
    multiset(const multiset&, const Allocator&);
    multiset(multiset&&, const Allocator&);
    multiset(initializer_list<value_type>,
```

§ 23.4.7.1 774

```
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& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;
// iterators:
iterator
                       begin() noexcept;
const_iterator
                       begin() const noexcept;
                       end() noexcept;
iterator
const_iterator
                       end() const noexcept;
reverse_iterator
                      rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
                       cbegin() const noexcept;
const iterator
                       cend() const noexcept;
const iterator
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;
// capacity:
bool
              empty() const noexcept;
              size() const noexcept;
size_type
              max_size() const noexcept;
size_type
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() noexcept;
// observers:
key_compare
             key_comp() const;
value_compare value_comp() const;
// set operations:
                find(const key_type& x);
iterator
const_iterator find(const key_type& x) const;
```

§ 23.4.7.1 775

```
size_type count(const key_type& x) const;
                         lower_bound(const key_type& x);
         iterator
         const_iterator lower_bound(const key_type& x) const;
                         upper_bound(const key_type& x);
         const_iterator upper_bound(const key_type& x) const;
         pair<iterator,iterator>
                                              equal_range(const key_type& x);
         pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
       template <class Key, class Compare, class Allocator>
         bool operator == (const multiset < Key, Compare, Allocator > & x,
                         const multiset<Key,Compare,Allocator>& y);
       template <class Key, class Compare, class Allocator>
         bool operator< (const multiset<Key,Compare,Allocator>& x,
                         const multiset<Key,Compare,Allocator>& y);
       template <class Key, class Compare, class Allocator>
         bool operator!=(const multiset<Key,Compare,Allocator>& x,
                         const multiset<Key,Compare,Allocator>& y);
       template <class Key, class Compare, class Allocator>
         bool operator> (const multiset<Key,Compare,Allocator>& x,
                          const multiset<Key,Compare,Allocator>& y);
       template <class Key, class Compare, class Allocator>
         bool operator>=(const multiset<Key,Compare,Allocator>& x,
                         const multiset<Key,Compare,Allocator>& y);
       template <class Key, class Compare, class Allocator>
         bool operator <= (const multiset < Key, Compare, Allocator > & x,
                          const multiset<Key,Compare,Allocator>& y);
       // specialized algorithms:
       template <class Key, class Compare, class Allocator>
         void swap(multiset<Key,Compare,Allocator>& x,
                   multiset<Key,Compare,Allocator>& y);
     }
   23.4.7.2 multiset constructors
                                                                                          [multiset.cons]
   explicit multiset(const Compare& comp = Compare(),
                     const Allocator& = Allocator());
        Effects: Constructs an empty set using the specified comparison object and allocator.
        Complexity: Constant.
   template <class InputIterator>
     multiset(InputIterator first, last,
              const Compare& comp = Compare(), const Allocator& = Allocator());
3
        Requires: If the iterator's dereference operator returns an Ivalue 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).
```

§ 23.4.7.2 776

Complexity: Linear in N if the range [first,last) is already sorted using comp and otherwise  $N \log N$ , where N is last - first.

#### 23.4.7.3 multiset specialized algorithms

[multiset.special]

#### 23.5 Unordered associative containers

[unord]

## 23.5.1 In general

[unord.general]

The header <unordered\_map> defines the class templates unordered\_map and unordered\_multimap; the header <unordered\_set> defines the class templates unordered\_set and unordered\_multiset.

## 23.5.2 Header <unordered\_map> synopsis

[unord.map.syn]

```
#include <initializer_list>
namespace std {
  // 23.5.4, class template unordered_map:
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<std::pair<const Key, T> > >
    class unordered_map;
  // 23.5.5, class template unordered_multimap:
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<std::pair<const Key, T> > >
    class unordered_multimap;
  template <class Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
              unordered_map<Key, T, Hash, Pred, Alloc>& y);
  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);
  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,
```

§ 23.5.2

```
const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
    template <class Key, class T, class Hash, class Pred, class Alloc>
     bool operator!=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                      const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
 } // namespace std
23.5.3 Header <unordered_set> synopsis
                                                                                   [unord.set.syn]
  #include <initializer_list>
 namespace std {
    // 23.5.6, class template unordered_set:
    template <class Key,
              class Hash = hash<Key>,
              class Pred = std::equal_to<Key>,
              class Alloc = std::allocator<Key> >
     class unordered_set;
    // 23.5.7, class template unordered multiset:
    template <class Key,
              class Hash = hash<Key>,
              class Pred = std::equal_to<Key>,
              class Alloc = std::allocator<Key> >
      class unordered_multiset;
    template <class Key, class Hash, class Pred, class Alloc>
      void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
                unordered_set<Key, Hash, Pred, Alloc>& y);
    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
                                                                                      [unord.map]
23.5.4 Class template unordered_map
                                                                             [unord.map.overview]
23.5.4.1 Class template unordered_map overview
```

- An unordered\_map is an unordered associative container that supports unique keys (an unordered\_map contains at most one of each key value) and that associates values of another type mapped\_type with the keys. The unordered\_map class supports forward iterators.
- An unordered\_map satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 99). It provides the operations described in the preceding requirements

§ 23.5.4.1 778

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>.

<sup>3</sup> This section only describes operations on unordered\_map that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
 template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Allocator = std::allocator<std::pair<const Key, T> > >
 class unordered_map
 public:
    // types
    typedef Key
                                                       key_type;
    typedef std::pair<const Key, T>
                                                       value_type;
    typedef T
                                                       mapped_type;
    typedef Hash
                                                       hasher;
    typedef Pred
                                                       key_equal;
    typedef Allocator
                                                       allocator_type;
    typedef typename allocator_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;
    \verb"typedef" implementation-defined"
                                                        const_iterator;
    {\tt typedef} \ \textit{implementation-defined}
                                                        local_iterator;
                                                        const_local_iterator;
    typedef implementation-defined
    // 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 1,
                    size_type n = see below,
                    const hasher& hf = hasher(),
                    const key_equal& eql = key_equal(),
                    const allocator_type& a = allocator_type());
    unordered_map(const unordered_map&);
    unordered_map(unordered_map&&);
    explicit unordered_map(const Allocator&);
    unordered_map(const unordered_map&, const Allocator&);
    unordered_map(unordered_map&&, const Allocator&);
    unordered_map(initializer_list<value_type>,
      size_type = see below,
      const hasher& hf = hasher(),
      const key_equal& eql = key_equal(),
      const allocator_type& a = allocator_type());
```

§ 23.5.4.1 779

```
~unordered map();
unordered_map& operator=(const unordered_map&);
unordered_map& operator=(unordered_map&&);
unordered_map& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;
// size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
// iterators
iterator
              begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
// modifiers
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& obj);
template <class P> pair<iterator, bool> insert(P&& obj);
              insert(const_iterator hint, const value_type& obj);
template <class P> iterator insert(const_iterator hint, P&& obj);
template <class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear() noexcept;
void swap(unordered_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 noexcept;
size_type max_bucket_count() const noexcept;
```

§ 23.5.4.1 780

```
size_type bucket_size(size_type n) const;
         size_type bucket(const key_type& k) const;
         local_iterator begin(size_type n);
         const_local_iterator begin(size_type n) const;
         local_iterator end(size_type n);
         const_local_iterator end(size_type n) const;
         const_local_iterator cbegin(size_type n) const;
         const_local_iterator cend(size_type n) const;
         // hash policy
         float load_factor() const noexcept;
         float max_load_factor() const noexcept;
         void max_load_factor(float z);
         void rehash(size_type n);
         void reserve(size_type n);
       };
       template <class Key, class 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>
         bool operator == (const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                         const unordered_map<Key, T, Hash, Pred, Alloc>& b);
       template <class Key, class T, class Hash, class Pred, class Alloc>
         bool operator!=(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                         const unordered_map<Key, T, Hash, Pred, Alloc>& b);
     }
   23.5.4.2
             unordered_map constructors
                                                                                     [unord.map.cnstr]
   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 func-
        tion, and allocator, and using at least n buckets. If n is not provided, the number of buckets is
        implementation-defined. max_load_factor() returns 1.0.
        Complexity: Constant.
   template <class InputIterator>
     unordered_map(InputIterator f, InputIterator 1,
                   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_map using the specified hash function, key equality func-
        tion, and allocator, and using at least n buckets. If n is not provided, the number of buckets is
        implementation-defined. Then inserts elements from the range [f, l). max_load_factor() returns
        Complexity: Average case linear, worst case quadratic.
```

§ 23.5.4.2 781

4

```
23.5.4.3 unordered_map element access
```

[unord.map.elem]

```
mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
```

1

1

6

Requires: mapped\_type shall be DefaultConstructible. For the first operator, key\_type shall be CopyConstructible. For the second operator, key\_type shall be MoveConstructible.

- 2 Effects: If the unordered\_map does not already contain an element whose key is equivalent to k, the first operator inserts the 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  $\mathcal{O}(1)$ , worst case  $\mathcal{O}(\text{size}())$ .

```
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.
- 6 Throws: An exception object of type out\_of\_range if no such element is present.

## 23.5.4.4 unordered\_map modifiers

[unord.map.modifiers]

```
template <class P>
  pair<iterator, bool> insert(P&& obj);
```

Requires: value\_type is constructible from std::forward<P>(obj).

- 2 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  $\mathcal{O}(1)$ , worst case  $\mathcal{O}(\text{size}())$ .
- 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.
- 8 Returns: An iterator that points to the element with key equivalent to the key of value\_type(obj).
- Complexity: Average case  $\mathcal{O}(1)$ , worst case  $\mathcal{O}(\text{size}())$ .
- Remarks: This signature shall not participate in overload resolution unless P is implicitly convertible to value\_type.

## 23.5.4.5 unordered\_map swap

[unord.map.swap]

```
template <class Key, class T, class Hash, class Pred, class Alloc> void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x, unordered_map<Key, T, Hash, Pred, Alloc>& y);

Effects: x.swap(y).
```

§ 23.5.4.5 782

#### 23.5.5 Class template unordered\_multimap

## [unord.multimap]

#### 23.5.5.1 Class template unordered\_multimap overview

[unord.multimap.overview]

An unordered\_multimap is an unordered associative container that supports equivalent keys (an instance of unordered\_multimap may contain multiple copies of each key value) and that associates values of another type mapped\_type with the keys. The unordered\_multimap class supports forward iterators.

- An unordered\_multimap satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 99). It provides the operations described in the preceding requirements table for equivalent keys; that is, an unordered\_multimap supports the a\_eq operations in that table, not the a\_uniq operations. For an unordered\_multimap<Key, T> the key type is Key, the mapped type is T, and the value type is std::pair<const Key, T>.
- <sup>3</sup> This section only describes operations on unordered\_multimap that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
 template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Allocator = std::allocator<std::pair<const Key, T> > >
 class unordered_multimap
  {
  public:
    // types
    typedef Key
                                                      key_type;
    typedef std::pair<const Key, T>
                                                      value_type;
    typedef T
                                                      mapped_type;
    typedef Hash
                                                      hasher;
    typedef Pred
                                                      key_equal;
    typedef Allocator
                                                      allocator_type;
    typedef typename allocator_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 = see below,
                                const hasher& hf = hasher(),
                                 const key_equal& eql = key_equal(),
                                 const allocator_type& a = allocator_type());
    template <class InputIterator>
      unordered_multimap(InputIterator f, InputIterator 1,
                         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&&);
```

§ 23.5.5.1 783

```
explicit 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>);
allocator_type get_allocator() const noexcept;
// size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
// iterators
iterator
               begin() noexcept;
const_iterator begin() const noexcept;
          end() noexcept;
iterator
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
// modifiers
template <class... Args> iterator emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
template <class P> iterator insert(P&& obj);
iterator insert(const_iterator hint, const value_type& obj);
template <class P> iterator insert(const_iterator hint, P&& obj);
template <class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear() noexcept;
void swap(unordered_multimap&);
// observers
hasher hash_function() const;
key_equal key_eq() const;
// lookup
               find(const key_type& k);
iterator
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
                                          equal_range(const key_type& k);
std::pair<iterator, iterator>
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
```

§ 23.5.5.1 784

```
// bucket interface
         size_type bucket_count() const noexcept;
         size_type max_bucket_count() const noexcept;
         size_type bucket_size(size_type n) const;
         size_type bucket(const key_type& k) const;
         local_iterator begin(size_type n);
         const_local_iterator begin(size_type n) const;
         local_iterator end(size_type n);
         const_local_iterator end(size_type n) const;
         const_local_iterator cbegin(size_type n) const;
         const_local_iterator cend(size_type n) const;
         // hash policy
         float load_factor() const noexcept;
         float max_load_factor() const noexcept;
         void max_load_factor(float z);
         void rehash(size_type n);
         void reserve(size_type n);
       };
       template <class Key, class 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);
       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);
     }
   23.5.5.2
             unordered_multimap constructors
                                                                                [unord.multimap.cnstr]
   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. max_load_factor() returns 1.0.
2
        Complexity: Constant.
   template <class InputIterator>
     unordered_multimap(InputIterator f, InputIterator 1,
                        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. Then inserts elements from the range [f, l). max_load_factor() returns
        1.0.
```

§ 23.5.5.2 785

4 Complexity: Average case linear, worst case quadratic.

```
23.5.5.3 unordered_multimap modifiers
```

[unord.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 \mathcal{O}(1), worst case \mathcal{O}(\text{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).
         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 \mathcal{O}(1), worst case \mathcal{O}(\text{size}()).
```

Remarks: This signature shall not participate in overload resolution unless P is implicitly convertible

#### 23.5.5.4 unordered\_multimap swap

to value\_type.

10

[unord.multimap.swap]

```
template <class Key, class T, class Hash, class Pred, class Alloc> void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x, unordered_multimap<Key, T, Hash, Pred, Alloc>& y);

Effects: x.swap(y).
```

#### 23.5.6 Class template unordered\_set

[unord.set]

#### 23.5.6.1 Class template unordered\_set overview

[unord.set.overview]

- An unordered\_set is an unordered associative container that supports unique keys (an unordered\_set contains at most one of each key value) and in which the elements' keys are the elements themselves. The unordered\_set class supports forward iterators.
- An unordered\_set satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 99). 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.
- 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.

§ 23.5.6.1 786

```
class unordered set
public:
 // types
 typedef Key
                                                    key_type;
  typedef Key
                                                    value_type;
  typedef Hash
                                                    hasher;
  typedef Pred
                                                    key_equal;
                                                    allocator_type;
  typedef Allocator
  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;
  {\tt typedef} \ \textit{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 = see below,
                         const hasher& hf = hasher(),
                         const key_equal& eql = key_equal(),
                         const allocator_type& a = allocator_type());
  template <class InputIterator>
    unordered_set(InputIterator f, InputIterator 1,
                  size_type n = see below,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
  unordered_set(const unordered_set&);
  unordered_set(unordered_set&&);
  explicit unordered_set(const Allocator&);
  unordered_set(const unordered_set&, const Allocator&);
  unordered_set(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 noexcept;
  // size and capacity
  bool empty() const noexcept;
  size_type size() const noexcept;
  size_type max_size() const noexcept;
  // iterators
  iterator
                 begin() noexcept;
```

§ 23.5.6.1 787

```
const_iterator begin() const noexcept;
            end() noexcept;
 iterator
  const_iterator end() const noexcept;
  const_iterator cbegin() const noexcept;
  const_iterator cend() const noexcept;
  // modifiers
  template <class... Args> pair<iterator, bool> emplace(Args&&... args);
  template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
  pair<iterator, bool> insert(const value_type& obj);
 pair<iterator, bool> insert(value_type&& obj);
 iterator insert(const_iterator hint, const value_type& obj);
  iterator insert(const_iterator hint, value_type&& obj);
  template <class InputIterator> void insert(InputIterator first, InputIterator last);
  void insert(initializer_list<value_type>);
 iterator erase(const_iterator position);
  size_type erase(const key_type& k);
  iterator erase(const_iterator first, const_iterator last);
  void clear() noexcept;
 void swap(unordered_set&);
  // observers
 hasher hash_function() const;
  key_equal key_eq() const;
  // lookup
  iterator
                 find(const key_type& k);
  const_iterator find(const key_type& k) const;
  size_type count(const key_type& k) const;
  std::pair<iterator, iterator>
                                            equal_range(const key_type& k);
  std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
  // bucket interface
  size_type bucket_count() const noexcept;
  size_type max_bucket_count() const noexcept;
  size_type bucket_size(size_type n) const;
  size_type bucket(const key_type& k) const;
  local_iterator begin(size_type n);
  const_local_iterator begin(size_type n) const;
  local_iterator end(size_type n);
  const_local_iterator end(size_type n) const;
  const_local_iterator cbegin(size_type n) const;
  const_local_iterator cend(size_type n) const;
  // hash policy
 float load_factor() const noexcept;
 float max_load_factor() const noexcept;
 void max_load_factor(float z);
 void rehash(size_type n);
  void reserve(size_type n);
template <class Key, class Hash, class Pred, class Alloc>
```

§ 23.5.6.1 788

```
void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
                   unordered_set<Key, Hash, Pred, Alloc>& y);
       template <class Key, class T, class Hash, class Pred, class Alloc>
         bool operator==(const unordered_set<Key, T, Hash, Pred, Alloc>& a,
                         const unordered_set<Key, T, Hash, Pred, Alloc>& b);
       template <class Key, class T, class Hash, class Pred, class Alloc>
         bool operator!=(const unordered set<Key, T, Hash, Pred, Alloc>& a,
                         const unordered_set<Key, T, Hash, Pred, Alloc>& b);
     }
   23.5.6.2
             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 func-
        tion, and allocator, and using at least n buckets. If n is not provided, the number of buckets is
        implementation-defined. max load factor() returns 1.0.
2
        Complexity: Constant.
   template <class InputIterator>
     unordered_set(InputIterator f, InputIterator 1,
                   size_type n = see below,
                   const hasher& hf = hasher(),
                   const key_equal& eql = key_equal(),
                   const allocator_type& a = allocator_type());
        Effects: Constructs an empty unordered_set using the specified hash function, key equality func-
        tion, and allocator, and using at least n buckets. If n is not provided, the number of buckets is
        implementation-defined. Then inserts elements from the range [f, l). max_load_factor() returns
        1.0.
4
        Complexity: Average case linear, worst case quadratic.
   23.5.6.3
                                                                                      [unord.set.swap]
             unordered_set swap
   template <class Key, class Hash, class Pred, class Alloc>
     void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
               unordered_set<Key, Hash, Pred, Alloc>& y);
        Effects: x.swap(y).
   23.5.7
            Class template unordered_multiset
                                                                                     [unord.multiset]
             Class template unordered_multiset overview
                                                                            [unord.multiset.overview]
```

- An unordered\_multiset is an unordered associative container that supports equivalent keys (an instance of unordered\_multiset may contain multiple copies of the same key value) and in which each element's key is the element itself. The unordered multiset class supports forward iterators.
- An unordered\_multiset satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 99). It provides the operations described in the preceding requirements table for equivalent keys; that is, an unordered\_multiset supports the a\_eq operations in that table, not the a\_uniq operations. For an unordered\_multiset<Key> the key type and the value type

§ 23.5.7.1 789

are both Key. The iterator and const\_iterator types are both const iterator types. It is unspecified whether they are the same type.

This section only describes operations on unordered\_multiset that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
 template <class Key,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Allocator = std::allocator<Key> >
  class unordered_multiset
 public:
    // types
   typedef Key
                                                       key_type;
    typedef Key
                                                       value_type;
    typedef Hash
                                                       hasher;
    typedef Pred
                                                       key_equal;
    typedef Allocator
                                                       allocator_type;
    typedef typename allocator_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;
    {\tt typedef} \ \textit{implementation-defined}
                                                       size_type;
    typedef implementation-defined
                                                       difference_type;
    typedef implementation-defined
                                                       iterator:
    typedef implementation-defined
                                                       const_iterator;
    {\tt typedef} \ \textit{implementation-defined}
                                                       local_iterator;
    typedef implementation-defined
                                                       const_local_iterator;
    // construct/destroy/copy
    explicit unordered_multiset(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_multiset(InputIterator f, InputIterator 1,
                         size_type n = see below,
                         const hasher& hf = hasher(),
                         const key_equal& eql = key_equal(),
                         const allocator_type& a = allocator_type());
    unordered_multiset(const unordered_multiset&);
    unordered_multiset(unordered_multiset&&);
    explicit unordered_multiset(const Allocator&);
    unordered_multiset(const unordered_multiset&, const Allocator&);
    unordered_multiset(unordered_multiset&&, const Allocator&);
    unordered_multiset(initializer_list<value_type>,
      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&&);
```

§ 23.5.7.1 790

```
unordered_multiset& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;
// size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
// iterators
iterator
              begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
// modifiers
template <class... Args> iterator emplace(Args&&... args);
template <class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template <class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear() noexcept;
void swap(unordered_multiset&);
// observers
hasher hash_function() const;
key_equal key_eq() const;
// lookup
iterator
               find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>
                                          equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;
```

§ 23.5.7.1 791

```
// hash policy
         float load_factor() const noexcept;
         float max_load_factor() const noexcept;
         void max_load_factor(float z);
         void rehash(size_type n);
         void reserve(size_type n);
       };
       template <class Key, class Hash, class Pred, class Alloc>
         void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
                   unordered_multiset<Key, Hash, Pred, Alloc>& y);
         template <class Key, class T, class Hash, class Pred, class Alloc>
           bool operator == (const unordered multiset < Key, T, Hash, Pred, Alloc > & a,
                           const unordered_multiset<Key, T, Hash, Pred, Alloc>& b);
         template <class Key, class T, class Hash, class Pred, class Alloc>
           bool operator!=(const unordered_multiset<Key, T, Hash, Pred, Alloc>& a,
                           const unordered_multiset<Key, T, Hash, Pred, Alloc>& b);
    }
   23.5.7.2
                                                                                 [unord.multiset.cnstr]
            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());
1
        Effects: Constructs an empty unordered_multiset using the specified hash function, key equality
        function, and allocator, and using at least n buckets. If n is not provided, the number of buckets is
        implementation-defined. max_load_factor() returns 1.0.
2
        Complexity: Constant.
   template <class InputIterator>
     unordered_multiset(InputIterator f, InputIterator 1,
                        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_multiset using the specified hash function, key equality
        function, and allocator, and using at least n buckets. If n is not provided, the number of buckets is
        implementation-defined. Then inserts elements from the range [f, l). max_load_factor() returns
        1.0.
        Complexity: Average case linear, worst case quadratic.
   23.5.7.3 unordered_multiset swap
                                                                                 [unord.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);
```

§ 23.5.7.3 792

## 23.6 Container adaptors

[container.adaptors]

#### 23.6.1 In general

[container.adaptors.general]

The headers <queue> and <stack> define the container adaptors queue, priority\_queue, and stack. These container adaptors meet the requirements for sequence containers.

- 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.
- <sup>3</sup> For container adaptors, no swap function throws an exception unless that exception is thrown by the swap of the adaptor's Container or Compare object (if any).

## 23.6.2 Header <queue> synopsis

[queue.syn]

```
#include <initializer_list>
namespace std {
  template <class T, class Container = deque<T> > class queue;
 template <class T, class Container = vector<T>,
    class Compare = less<typename Container::value_type> >
      class priority_queue;
  template <class T, class Container>
    bool operator == (const queue < T, Container > & x, const queue < T, Container > & y);
  template <class T, class Container>
    bool operator< (const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, class Container>
    bool operator!=(const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, class Container>
    bool operator> (const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, class Container>
    bool operator>=(const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, class Container>
    bool operator<=(const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, class Container>
    void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));
  template <class T, class Container, class Compare>
    void swap(priority_queue<T, Container, Compare>& x,
              priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));
}
```

#### 23.6.3 Class template queue

[queue]

#### 23.6.3.1 queue definition

[queue.defn]

Any sequence container supporting operations front(), back(), push\_back() and pop\_front() can be used to instantiate queue. In particular, list (23.3.5) and deque (23.3.3) can be used.

```
namespace std {
  template <class T, class Container = deque<T> >
  class queue {
  public:
    typedef typename Container::value_type value_type;
    typedef typename Container::reference reference;
    typedef typename Container::const_reference const_reference;
    typedef typename Container::size_type size_type;
```

§ 23.6.3.1 793

container\_type;

Container

1

```
typedef
   protected:
     Container c;
   public:
      explicit queue(const Container&);
      explicit queue(Container&& = Container());
      template <class Alloc> explicit queue(const Alloc&);
      template <class Alloc> queue(const Container&, const Alloc&);
      template <class Alloc> queue(Container&&, const Alloc&);
      template <class Alloc> queue(const queue&, const Alloc&);
      template <class Alloc> queue(queue&&, const Alloc&);
     bool
                       empty() const
                                         { return c.empty(); }
                       size() const
                                        { return c.size(); }
     size_type
                      front()
                                        { return c.front(); }
     reference
     const_reference front() const { return c.front(); }
                      back()
     reference
                                       { return c.back(); }
     const_reference back() const { return c.back(); }
      void push(const value_type& x)
                                       { c.push_back(x); }
      void push(value_type&& x)
                                         { c.push_back(std::move(x)); }
      template <class... Args> void emplace(Args&&... args)
        { c.emplace_back(std::forward<Args>(args)...); }
     void pop()
                                         { c.pop_front(); }
     void swap(queue& q) noexcept(noexcept(swap(c, q.c)))
        { using std::swap; swap(c, q.c); }
   };
   template <class T, class Container>
     bool operator == (const queue < T, Container > & x, const queue < T, Container > & y);
   template <class T, class Container>
     bool operator< (const queue<T, Container>& x, const queue<T, Container>& y);
   template <class T, class Container>
     bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);
   template <class T, class Container>
     bool operator> (const queue<T, Container>& x, const queue<T, Container>& y);
   template <class T, class Container>
     bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);
   template <class T, class Container>
     bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);
   template <class T, class Container>
     void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));
   template <class T, class Container, class Alloc>
     struct uses_allocator<queue<T, Container>, Alloc>
        : uses_allocator<Container, Alloc>::type { };
 }
23.6.3.2 queue constructors
                                                                                      [queue.cons]
explicit queue(const Container& cont);
     Effects: Initializes c with cont.
explicit queue(Container&& cont = Container());
§ 23.6.3.2
                                                                                               794
```

2 Effects: Initializes c with std::move(cont). 23.6.3.3 queue constructors with allocators [queue.cons.alloc] If uses\_allocator<container\_type, Alloc>::value is false the constructors in this subclause shall not participate in overload resolution. template <class Alloc> explicit queue(const Alloc& a); Effects: Initializes c with a. template <class Alloc> queue(const container\_type& cont, const Alloc& a); 3 Effects: Initializes c with cont as the first argument and a as the second argument. template <class Alloc> queue(container\_type&& cont, const Alloc& a); Effects: Initializes c with std::move(cont) as the first argument and a as the second argument. template <class Alloc> queue(const queue& q, const Alloc& a); 5 Effects: Initializes c with q.c as the first argument and a as the second argument. template <class Alloc> queue(queue&& q, const Alloc& a); Effects: Initializes c with std::move(q.c) as the first argument and a as the second argument. 23.6.3.4queue operators [queue.ops] template <class T, class Container> bool operator==(const queue<T, Container>& x, const queue<T, Container>& y); Returns: x.c == y.c.template <class T, class Container> bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y); Returns: x.c != y.c.template <class T, class Container> bool operator< (const queue<T, Container>& x, const queue<T, Container>& y); 3 Returns: x.c < y.c.

§ 23.6.3.4 795

```
template <class T, class Container>
       bool operator<=(const queue<T, Container>& x,
                       const queue<T, Container>& y);
4
        Returns: x.c \le 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);
        Returns: x.c >= y.c.
             queue specialized algorithms
   23.6.3.5
                                                                                        [queue.special]
   template <class T, class Container>
     void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));
        Effects: x.swap(y).
   23.6.4 Class template priority_queue
                                                                                      [priority.queue]
  Any sequence container with random access iterator and supporting operations front(), push_back() and
   pop_back() can be used to instantiate priority_queue. In particular, vector (23.3.6) and deque (23.3.3)
   can be used. Instantiating priority_queue also involves supplying a function or function object for mak-
   ing 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,
```

§ 23.6.4 796

```
const Compare& x = Compare(), Container&& = Container());
         template <class Alloc> explicit priority_queue(const Alloc&);
         template <class Alloc> priority_queue(const Compare&, const Alloc&);
         template <class Alloc> priority_queue(const Compare&,
           const Container&, const Alloc&);
         template <class Alloc> priority_queue(const Compare&,
           Container&&, const Alloc&);
         template <class Alloc> priority_queue(const priority_queue&, const Alloc&);
         template <class Alloc> priority_queue(priority_queue&&, const Alloc&);
                   empty() const
                                        { return c.empty(); }
         size_type size() const
                                        { return c.size(); }
         const_reference
                          top() const { return c.front(); }
         void push(const value_type& x);
         void push(value_type&& x);
         template <class... Args> void emplace(Args&&... args);
         void pop();
         void swap(priority_queue& q) noexcept(
             noexcept(swap(c, q.c)) && noexcept(swap(comp, q.comp)))
           { using std::swap; swap(c, q.c); swap(comp, q.comp); }
       };
       // no equality is provided
       template <class T, class Container, class Compare>
         void swap(priority_queue<T, Container, Compare>& x,
                   priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));
       template <class T, class Container, class Compare, class Alloc>
         struct uses_allocator<priority_queue<T, Container, Compare>, Alloc>
           : uses_allocator<Container, Alloc>::type { };
     }
   23.6.4.1 priority_queue constructors
                                                                                        [priqueue.cons]
   priority_queue(const Compare& x,
                  const Container& y);
   explicit priority_queue(const Compare& x = Compare(),
                  Container&& y = Container());
        Requires: x shall define a strict weak ordering (25.4).
        Effects: Initializes comp with x and c with y (copy constructing or move constructing as appropriate);
        calls make_heap(c.begin(), c.end(), comp).
   template <class InputIterator>
     priority_queue(InputIterator first, InputIterator last,
                    const Compare& x,
                    const Container& y);
   template <class InputIterator>
     priority_queue(InputIterator first, InputIterator last,
                    const Compare& x = Compare(),
                    Container&& y = Container());
3
        Requires: x shall define a strict weak ordering (25.4).
        Effects: Initializes comp with x and c with y (copy constructing or move constructing as appropriate);
        calls c.insert(c.end(), first, last); and finally calls make heap(c.begin(), c.end(), comp).
```

1

4

§ 23.6.4.1 797

[priqueue.cons.alloc]

23.6.4.2 priority\_queue 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 priority_queue(const Alloc& a);
        Effects: Initializes c with a and value-initializes comp.
   template <class Alloc>
     priority_queue(const Compare& compare, const Alloc& a);
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);
        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);
        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);
         Effects: Initializes c with q.c as the first argument and a as the second argument, and initializes comp
        with q.comp.
   template <class Alloc>
     priority_queue(priority_queue&& q, const Alloc& a);
         Effects: Initializes c with std::move(q.c) as the first argument and a as the second argument, and
        initializes comp with std::move(q.comp).
                                                                                     [priqueue.members]
   23.6.4.3 priority_queue members
   void push(const value_type& x);
        Effects:
          c.push_back(x);
          push_heap(c.begin(), c.end(), comp);
   void push(value_type&& x);
2
         Effects:
          c.push_back(std::move(x));
          push_heap(c.begin(), c.end(), comp);
   § 23.6.4.3
                                                                                                        798
```

```
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();
   23.6.4.4 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) noexcept(noexcept(x.swap(y)));
        Effects: x.swap(y).
   23.6.5 Class template stack
                                                                                                [stack]
  Any sequence container supporting operations back(), push_back() and pop_back() can be used to in-
   stantiate stack. In particular, vector (23.3.6), list (23.3.5) and deque (23.3.3) can be used.
   23.6.5.1 Header <stack> synopsis
                                                                                            [stack.syn]
     #include <initializer_list>
    namespace std {
       template <class T, class Container = deque<T> > class stack;
       template <class T, class Container>
         bool operator==(const stack<T, Container>& x,const stack<T, Container>& y);
       template <class T, class Container>
        bool operator< (const stack<T, Container>& x,const stack<T, Container>& y);
       template <class T, class Container>
         bool operator!=(const stack<T, Container>& x,const stack<T, Container>& y);
       template <class T, class Container>
         bool operator> (const stack<T, Container>& x,const stack<T, Container>& y);
      template <class T, class Container>
         bool operator>=(const stack<T, Container>& x,const stack<T, Container>& y);
       template <class T, class Container>
         bool operator<=(const stack<T, Container>& x,const stack<T, Container>& y);
       template <class T, class Container>
         void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y)));
     }
   23.6.5.2 stack definition
                                                                                           [stack.defn]
    namespace std {
      template <class T, class Container = deque<T> >
       class stack {
       public:
         typedef typename Container::value_type
                                                           value_type;
   § 23.6.5.2
                                                                                                    799
```

reference;

typedef typename Container::reference

```
typedef typename Container::const_reference
                                                        const_reference;
      typedef typename Container::size_type
                                                        size_type;
                       Container
      typedef
                                                        container_type;
    protected:
      Container c;
   public:
      explicit stack(const Container&);
      explicit stack(Container&& = Container());
      template <class Alloc> explicit stack(const Alloc&);
      template <class Alloc> stack(const Container&, const Alloc&);
      template <class Alloc> stack(Container&&, const Alloc&);
      template <class Alloc> stack(const stack&, const Alloc&);
      template <class Alloc> stack(stack&&, const Alloc&);
     bool
                empty() const
                                          { return c.empty(); }
     size_type size() const
                                          { return c.size(); }
                        top()
                                          { return c.back(); }
     reference
      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)...); }
                                          { c.pop_back(); }
      void pop()
     void swap(stack& s) noexcept(noexcept(swap(c, s.c)))
        { using std::swap; swap(c, s.c); }
    };
    template <class T, class Container>
     bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);
    template <class T, class Container>
     bool operator< (const stack<T, Container>& x, const stack<T, Container>& y);
   template <class T, class Container>
     bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);
    template <class T, class Container>
     bool operator> (const stack<T, Container>& x, const stack<T, Container>& y);
    template <class T, class Container>
     bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);
    template <class T, class Container>
     bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);
    template <class T, class Allocator>
     void swap(stack<T,Allocator>& x, stack<T,Allocator>& y);
    template <class T, class Container, class Alloc>
     struct uses_allocator<stack<T, Container>, Alloc>
        : uses_allocator<Container, Alloc>::type { };
23.6.5.3 stack constructors
                                                                                        [stack.cons]
explicit stack(const Container& cont);
     Effects: Initializes c with cont.
explicit stack(Container&& const = Container());
§ 23.6.5.3
                                                                                                 800
```

2 Effects: Initializes c with std::move(cont). 23.6.5.4 stack constructors with allocators [stack.cons.alloc] If uses\_allocator<container\_type, Alloc>::value is false the constructors in this subclause shall not participate in overload resolution. template <class Alloc> explicit stack(const Alloc& a); Effects: Initializes c with a. template <class Alloc> stack(const container\_type& cont, const Alloc& a); 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(cont) as the first argument and a as the second argument. template <class Alloc> stack(const stack& s, const Alloc& a); 5 Effects: Initializes c with s.c as the first argument and a as the second argument. template <class Alloc> stack(stack&& s, const Alloc& a); Effects: Initializes c with std::move(s.c) as the first argument and a as the second argument. 23.6.5.5stack 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); 3 Returns: x.c < y.c.

§ 23.6.5.5

```
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);
5
        Returns: x.c > y.c.
   template <class T, class Container>
       bool operator>=(const stack<T, Container>& x,
                       const stack<T, Container>& y);
        Returns: x.c >= y.c.
   23.6.5.6 stack specialized algorithms
                                                                                        [stack.special]
   template <class T, class Container>
     void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y)));
        Effects: x.swap(y).
```

§ 23.6.5.6 802

## 24 Iterators library

# [iterators]

24.1 General [iterators.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).

<sup>2</sup> The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 104.

Table 104 —	Iterators	library	summary

	Subclause	Header(s)
24.2	Requirements	
24.4	Iterator primitives	<iterator></iterator>
24.5	Predefined iterators	
24.6	Stream iterators	

## 24.2 Iterator requirements

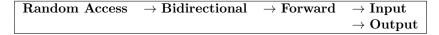
## [iterator.requirements]

## 24.2.1 In general

## [iterator.requirements.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 object type T, called the value type of the iterator. All output iterators support the expression \*i = o where o is a value of some type that is in the set of types that are writable to the particular iterator type of i. All iterators i for which the expression (\*i).m is well-defined, support the expression i->m with the same semantics as (\*i).m. For every iterator type X for which equality is defined, there is a corresponding signed integer type called the difference type of the iterator.
- Since iterators are an abstraction of pointers, their semantics is a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This International Standard defines five categories of iterators, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*, as shown in Table 105.

Table 105 — Relations among iterator categories



- <sup>3</sup> 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.
- <sup>4</sup> Iterators that further satisfy the requirements of output iterators are called *mutable iterators*. Nonmutable iterators are referred to as *constant iterators*.

§ 24.2.1

Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called past-the-end values. Values of an iterator i for which the expression \*i is defined are called dereferenceable. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any sequence. [Example: After the declaration of an uninitialized pointer x (as with int\* x;), x must always be assumed to have a singular value of a pointer. —end example] Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value, the assignment of a non-singular value to an iterator that holds a singular value, and, for iterators that satisfy the DefaultConstructible requirements, using a value-initialized iterator as the source of a copy or move operation. [Note: This guarantee is not offered for default initialization, although the distinction only matters for types with trivial default constructors such as pointers or aggregates holding pointers. —end note] In these cases the singular value is overwritten the same way as any other value. Dereferenceable values are always non-singular.

- An iterator j is called *reachable* from an iterator i if and only if there is a finite sequence of applications of the expression ++i that makes i == j. If j is reachable from i, they refer to elements of the same sequence.
- Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A range is a pair of iterators that designate the beginning and end of the computation. A range [i,i) is an empty range; in general, a range [i,j) refers to the elements in the data structure starting with the element pointed to by i and up to but not including the element pointed to by j. Range [i,j) is valid if and only if j is reachable from i. The result of the application of functions in the library to invalid ranges is undefined.
- <sup>8</sup> 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.
- 9 Destruction of an iterator may invalidate pointers and references previously obtained from that iterator.
- $^{10}$  An *invalid* iterator is an iterator that may be singular.  $^{267}$
- In the following sections, a and b denote values of type X or const X, difference\_type and reference refer to the types iterator\_traits<X>::difference\_type and iterator\_traits<X>::reference, respectively, n denotes a value of difference\_type, u, tmp, and m denote identifiers, r denotes a value of X&, t denotes a value of value type T, o denotes a value of some type that is writable to the output iterator. [Note: For an iterator type X there must be an instantiation of iterator\_traits<X> (24.4.1). end note]

## 24.2.2 Iterator [iterator.iterators]

- The Iterator requirements form the basis of the iterator concept taxonomy; every iterator satisfies the Iterator requirements. This set of requirements specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to read (24.2.3) or write (24.2.4) values, or to provide a richer set of iterator movements (24.2.5, 24.2.6, 24.2.7).)
- A type X satisfies the Iterator requirements if:
  - X satisfies the CopyConstructible, CopyAssignable, and Destructible requirements (17.6.3.1) and lvalues of type X are swappable (17.6.3.2), and
  - the expressions in Table 106 are valid and have the indicated semantics.

Table 106 — Iterator requirements

Expression	Return type	Operational semantics	$\begin{array}{c} {\bf Assertion/note} \\ {\bf pre-/post-condition} \end{array}$
*r	reference		pre: $\mathbf{r}$ is dereferenceable.
++r	X&		

<sup>267)</sup> This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.

§ 24.2.2

## 24.2.3 Input iterators

## [input.iterators]

A class or pointer type X satisfies the requirements of an input iterator for the value type T if X satisfies the Iterator (24.2.2) and EqualityComparable (Table 17) requirements and the expressions in Table 107 are valid and have the indicated semantics.

In Table 107, the term the domain of == is used in the ordinary mathematical sense to denote the set of values over which == is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of == for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of == and !=. [Example: the call find(a,b,x) is defined only if the value of a has the property p defined as follows: b has property p and a value i has property p if (\*i==x) or if (\*i!=x and ++i has property p). — end example]

Expression	Return type	Operational semantics	$egin{aligned} & \mathbf{Assertion/note} \ & \mathbf{pre-/post-condition} \end{aligned}$
a != b	contextually convertible to bool	!(a == b)	pre: $(a, b)$ is in the domain of $==$ .
*a	convertible to T		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.
a->m		(*a).m	pre: a is dereferenceable.
++r	X&		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 ==.
(void)r++			equivalent to (void)++r
*r++	convertible to T	{ T tmp = *r; ++r; return tmp; }	

Table 107 — Input iterator requirements (in addition to Iterator)

[Note: For input iterators, a == b does not imply ++a == ++b. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms. Value type T is not required to be a CopyAssignable type (Table 23). These algorithms can be used with istreams as the source of the input data through the istream\_iterator class template. — end note]

## 24.2.4 Output iterators

[output.iterators]

A class or pointer type X satisfies the requirements of an output iterator if X satisfies the Iterator requirements (24.2.2) and the expressions in Table 108 are valid and have the indicated semantics.

§ 24.2.4 805

Expression	Return type	Operational	Assertion/note
_		semantics	pre-/post-condition
*r = 0	result is not		Remark: After this operation
	used		<b>r</b> is not required to be
			dereferenceable.
			post: r is incrementable.
++r	X&		&r == &++r.
			Remark: After this operation
			r is not required to be
			dereferenceable.
			post: r is incrementable.
r++	convertible to	{ X tmp = r;	Remark: After this operation
	const X&	++r;	r is not required to be
		return tmp; }	dereferenceable.
			post: r is incrementable.
*r++ = 0	result is not		Remark: After this operation
	used		r is not required to be
			dereferenceable.
			post: r is incrementable.

Table 108 — Output iterator requirements (in addition to Iterator)

[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

[forward.iterators]

- A class or pointer 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 (17.6.3.1),
  - 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 109 are valid and have the indicated semantics, and
  - objects of type X offer the multi-pass guarantee, described below.
- <sup>2</sup> The domain of == for forward iterators is that of iterators over the same underlying sequence.
- 3 Two dereferenceable iterators a and b of type X offer the multi-pass quarantee if:
  - a == b implies ++a == ++b and
  - X is a pointer type or the expression (void)++X(a), \*a is equivalent to the expression \*a.
- <sup>4</sup> [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]

§ 24.2.5

Expression	Return type	Operational semantics	$\begin{array}{c} \textbf{Assertion/note} \\ \textbf{pre-/post-condition} \end{array}$
r++	convertible to	{ X tmp = r;	
	const X&	++r;	
		<pre>return tmp; }</pre>	
*r++	reference		

Table 109 — Forward iterator requirements (in addition to input iterator)

- 5 If a and b are equal, then either a and b are both dereferenceable or else neither is dereferenceable.
- <sup>6</sup> 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

## [bidirectional.iterators]

A class or pointer type X satisfies the requirements of a bidirectional iterator if, in addition to satisfying the requirements for forward iterators, the following expressions are valid as shown in Table 110.

Table 110 — Bidirectional iterator requirements (in addition to forward iterator)

Expression	Return type	Operational semantics	$egin{aligned} & \mathbf{Assertion/note} \ & \mathbf{pre-/post-condition} \end{aligned}$
r	X&		pre: there exists s such that  r == ++s.  post: r is dereferenceable. (++r) == r. r ==s implies r == s.  &r == &r.
r	convertible to	{ X tmp = r; r; return tmp; }	
*r	reference		

<sup>2</sup> [Note: Bidirectional iterators allow algorithms to move iterators backward as well as forward. — end note]
24.2.7 Random access iterators [random.access.iterators]

A class or pointer type X satisfies the requirements of a random access iterator if, in addition to satisfying the requirements for bidirectional iterators, the following expressions are valid as shown in Table 111.

§ 24.2.7 807

Table 111 — Random access iterator requirements (in addition to bidirectional iterator)

Expression	Return type	$egin{array}{c} \mathbf{Operational} \ \mathbf{semantics} \end{array}$	$\begin{array}{c} \textbf{Assertion/note} \\ \textbf{pre-/post-condition} \end{array}$
r += n	X&	{ difference_type m = n; if (m >= 0)	
		while (m)	
		++r;	
		else	
		while (m++)	
		r;	
		return r; }	
a + n	X	{ X tmp = a;	a + n == n + a.
n + a		return tmp += n; }	
r -= n	X&	return r += -n;	
a - n	Х	{ X tmp = a;	
		return tmp -= n; }	
b - a	difference	return n	pre: there exists a value n of
	type		$\operatorname{type}$ difference_type $\operatorname{such}$
			that $a + n == b$ .
			b == a + (b - a).
a[n]	convertible to	*(a + n)	
	reference		
a < b	contextually	b - a > 0	< is a total ordering relation
	convertible to		
	bool		
a > b	contextually	b < a	> is a total ordering relation
	convertible to		opposite to <.
	bool		
a >= b	contextually	!(a < b)	
	convertible to		
	bool		
a <= b	contextually	!(a > b)	
	convertible to		
	bool.		

## 24.3 Header <iterator> synopsis

[iterator.synopsis]

§ 24.3

```
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<(</pre>
    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<=(</pre>
    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>
```

§ 24.3

```
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<=(</pre>
    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_iteratoriterator2>& y) -> decltype(x.base() - y.base());
template <class Iterator>
  move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);
template <class Iterator>
  move_iterator<Iterator> make_move_iterator(const Iterator& i);
// 24.6, stream iterators:
template <class T, class charT = char, class traits = char_traits<charT>,
    class Distance = ptrdiff_t>
class istream_iterator;
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);
template <class T, class charT, class traits, class Distance>
  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> >
```

§ 24.3

```
class istreambuf_iterator;
  template <class charT, class traits>
    bool operator == (const istreambuf_iterator < charT, traits > & a,
            const istreambuf_iterator<charT,traits>& b);
 template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);
  template <class charT, class traits = char_traits<charT> >
    class ostreambuf_iterator;
  // 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

[iterator.primitives]

To simplify the task of defining iterators, the library provides several classes and functions:

#### 24.4.1 Iterator traits

[iterator.traits]

To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if Iterator is the type of an iterator, the types

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::iterator_category
```

be defined as the iterator's difference type, value type and iterator category, respectively. In addition, the types

```
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

shall be defined as the iterator's reference and pointer types, that is, for an iterator object a, the same type as the type of \*a and a->, respectively. In the case of an output iterator, the types

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

may be defined as void.

2 The template iterator\_traits<Iterator> is defined as

```
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;
};
}
```

§ 24.4.1 811

```
<sup>3</sup> It is specialized for pointers as
     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
     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;
       };
     }
<sup>4</sup> [Note: If there is an additional pointer type _ far such that the difference of two _ far is of type long,
   an implementation may define
       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]
<sup>5</sup> [Example: To implement a generic reverse function, a C++ program can do the following:
     template <class BidirectionalIterator>
     void reverse(BidirectionalIterator first, BidirectionalIterator last) {
       typename iterator_traits<BidirectionalIterator>::difference_type n =
         distance(first, last);
       --n;
       while(n > 0) {
         typename iterator_traits<BidirectionalIterator>::value_type
          tmp = *first;
         *first++ = *--last;
         *last = tmp;
         n = 2;
     }
   — end example]
   24.4.2 Basic iterator
                                                                                          [iterator.basic]
 The iterator template may be used as a base class to ease the definition of required types for new iterators.
     namespace std {
       template<class Category, class T, class Distance = ptrdiff_t,</pre>
   § 24.4.2
                                                                                                        812
```

## 24.4.3 Standard iterator tags

[std.iterator.tags]

It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces category tag classes which are used as compile time tags for algorithm selection. They are: input\_iterator\_tag, output\_iterator\_tag, forward\_iterator\_tag, bidirectional\_iterator\_tag and random\_access\_iterator\_tag. For every iterator of type Iterator, iterator\_traits<Iterator>::iterator\_category shall be defined to be the most specific category tag that describes the iterator's behavior.

```
namespace std {
   struct input_iterator_tag { };
   struct output_iterator_tag { };
   struct forward_iterator_tag: public input_iterator_tag { };
   struct bidirectional_iterator_tag: public forward_iterator_tag { };
   struct random_access_iterator_tag: public bidirectional_iterator_tag { };
}
```

[Example: For a program-defined iterator BinaryTreeIterator, it could be included into the bidirectional iterator category by specializing the iterator\_traits template:

```
template<class T> struct iterator_traits<BinaryTreeIterator<T> > {
  typedef std::ptrdiff_t difference_type;
  typedef T value_type;
  typedef T* pointer;
  typedef T& reference;
  typedef bidirectional_iterator_tag iterator_category;
};
```

Typically, however, it would be easier to derive BinaryTreeIterator<T> from iterator<br/>bidirectional\_-iterator\_tag,T,ptrdiff\_t,T\*,T&>. — end example |

<sup>3</sup> [Example: If evolve() is well defined for bidirectional iterators, but can be implemented more efficiently for random access iterators, then the implementation is as follows:

```
template <class BidirectionalIterator>
inline void
evolve(BidirectionalIterator first, BidirectionalIterator last) {
   evolve(first, last,
        typename iterator_traits<BidirectionalIterator>::iterator_category());
}

template <class BidirectionalIterator>
void evolve(BidirectionalIterator first, BidirectionalIterator last,
   bidirectional_iterator_tag) {
   // more generic, but less efficient algorithm
}
```

§ 24.4.3

```
template <class RandomAccessIterator>
     void evolve(RandomAccessIterator first, RandomAccessIterator last,
       random_access_iterator_tag) {
       // more efficient, but less generic algorithm
   — end example]
<sup>4</sup> [Example: If a C++ program wants to define a bidirectional iterator for some data structure containing
   double and such that it works on a large memory model of the implementation, it can do so with:
     class MyIterator :
       public iterator<bidirectional_iterator_tag, double, long, T*, T&> {
       // code implementing ++, etc.
     };
  Then there is no need to specialize the iterator traits template. — end example]
   24.4.4 Iterator operations
                                                                                   [iterator.operations]
<sup>1</sup> Since only random access iterators provide + and - operators, the library provides two function templates
   advance and distance. These function templates use + and - for random access iterators (and are, therefore,
   constant time for them); for input, forward and bidirectional iterators they use ++ to provide linear time
   implementations.
   template <class InputIterator, class Distance>
     void advance(InputIterator& i, Distance n);
2
        Requires: n shall be negative only for bidirectional and random access iterators.
3
        Effects: Increments (or decrements for negative n) iterator reference i by n.
     template < class InputIterator>
         typename iterator_traits<InputIterator>::difference_type
            distance(InputIterator first, InputIterator last);
4
        Effects: If InputIterator meets the requirements of random access iterator, returns (last - first);
        otherwise, returns the number of increments needed to get from first to last.
5
        Requires: If InputIterator meets the requirements of random access iterator, last shall be reachable
        from first or first shall be reachable from last; otherwise, last shall be reachable from first.
   template <class ForwardIterator>
     ForwardIterator next(ForwardIterator x,
       typename std::iterator_traits<ForwardIterator>::difference_type n = 1);
6
        Effects: Equivalent to advance(x, n); return x;
   template <class BidirectionalIterator>
     BidirectionalIterator prev(BidirectionalIterator x,
       typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
        Effects: Equivalent to advance(x, -n); return x;
```

§ 24.4.4 814

## 24.5 Iterator adaptors

## [predef.iterators]

#### 24.5.1 Reverse iterators

[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

[reverse.iterator]

```
namespace std {
  template <class Iterator>
 class reverse_iterator : public
        iterator<typename iterator_traits<Iterator>::iterator_category,
        typename iterator_traits<Iterator>::value_type,
        typename iterator_traits<Iterator>::difference_type,
        typename iterator_traits<Iterator>::pointer,
        typename iterator_traits<Iterator>::reference> {
  public:
    typedef Iterator
                                                                 iterator_type;
    typedef typename iterator_traits<Iterator>::difference_type difference_type;
    typedef typename iterator_traits<Iterator>::reference
                                                                 reference;
    typedef typename iterator_traits<Iterator>::pointer
                                                                 pointer;
    reverse_iterator();
    explicit reverse_iterator(Iterator x);
    template <class U> reverse_iterator(const reverse_iterator<U>& u);
    template <class U> reverse_iterator& operator=(const reverse_iterator<U>& u);
                                // explicit
    Iterator base() const;
    reference operator*() const;
    pointer operator->() const;
    reverse_iterator& operator++();
    reverse_iterator operator++(int);
    reverse_iterator& operator--();
    reverse_iterator operator--(int);
    reverse_iterator operator+ (difference_type n) const;
    reverse_iterator& operator+=(difference_type n);
    reverse_iterator operator- (difference_type n) const;
    reverse_iterator& operator-=(difference_type n);
    unspecified operator[](difference_type n) const;
  protected:
    Iterator current;
  private:
                                // exposition only
    Iterator deref_tmp;
  template <class Iterator1, class Iterator2>
    bool operator==(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
 template <class Iterator1, class Iterator2>
    bool operator<(</pre>
      const reverse_iterator<Iterator1>& x,
```

§ 24.5.1.1 815

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<=(</pre>
           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
                                                                              [reverse.iter.requirements]
  The template parameter Iterator shall meet all the requirements of a Bidirectional Iterator (24.2.6).
  Additionally, Iterator shall meet the requirements of a Random Access Iterator (24.2.7) if any of the mem-
   bers 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 \leq (24.5.1.3.18), operator \geq (24.5.1.3.17), operator \leq (24.5.1.3.19) or operator \leq (24.5.1.3.20) are
   referenced in a way that requires instantiation (14.7.1).
   24.5.1.3 reverse_iterator operations
                                                                                         [reverse.iter.ops]
   24.5.1.3.1 reverse_iterator constructor
                                                                                        [reverse.iter.cons]
   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);
         Effects: Initializes current with x.
   template <class U> reverse_iterator(const reverse_iterator<U> &u);
3
        Effects: Initializes current with u.current.
```

§ 24.5.1.3.1

```
[reverse.iter.op=]
   24.5.1.3.2 reverse_iterator::operator=
   template <class U>
   reverse_iterator&
     operator=(const reverse_iterator<U>& u);
        Effects: Assigns u.base() to current.
2
        Returns: *this.
   24.5.1.3.3 Conversion
                                                                                     [reverse.iter.conv]
   Iterator base() const;
                                   // explicit
        Returns: current.
   24.5.1.3.4 operator*
                                                                                   [reverse.iter.op.star]
   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->
                                                                                     [reverse.iter.opref]
   pointer operator->() const;
        Returns: &(operator*()).
                                                                                    [reverse.iter.op++]
   24.5.1.3.6 operator++
  reverse_iterator& operator++();
1
        Effects: --current;
2
        Returns: *this.
   reverse_iterator operator++(int);
3
        \it Effects:
          reverse_iterator tmp = *this;
          --current;
          return tmp;
                                                                                      [reverse.iter.op--]
   24.5.1.3.7
              operator--
   reverse_iterator& operator--();
        Effects: ++current
1
2
        Returns: *this.
  reverse_iterator operator--(int);
                                                                                                     817
   § 24.5.1.3.7
```

```
3
        Effects:
          reverse_iterator tmp = *this;
          ++current;
          return tmp;
   24.5.1.3.8
               operator+
                                                                                    [reverse.iter.op+]
   reverse_iterator
   operator+(typename reverse_iterator<Iterator>::difference_type n) const;
        Returns: reverse_iterator(current-n).
   24.5.1.3.9
               operator+=
                                                                                  [reverse.iter.op+=]
   reverse_iterator&
   operator+=(typename reverse_iterator<Iterator>::difference_type n);
        Effects: current -= n;
        Returns: *this.
   24.5.1.3.10 operator-
                                                                                     [reverse.iter.op-]
  reverse_iterator
   operator-(typename reverse_iterator<Iterator>::difference_type n) const;
        Returns: reverse_iterator(current+n).
                                                                                   [reverse.iter.op-=]
   24.5.1.3.11 operator-=
   reverse_iterator&
   operator-=(typename reverse_iterator<Iterator>::difference_type n);
        Effects: current += n;
        Returns: *this.
   24.5.1.3.12 operator[]
                                                                                [reverse.iter.opindex]
   unspecified operator[](
       typename reverse_iterator<Iterator>::difference_type n) const;
        Returns: current[-n-1].
   24.5.1.3.13 operator==
                                                                                  [reverse.iter.op==]
   template <class Iterator1, class Iterator2>
    bool operator==(
       const reverse_iterator<Iterator1>& x,
       const reverse_iterator<Iterator2>& y);
        Returns: x.current == y.current.
   24.5.1.3.14 operator<
                                                                                    [reverse.iter.op<]
   template <class Iterator1, class Iterator2>
    bool operator<(</pre>
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
        Returns: x.current > y.current.
```

§ 24.5.1.3.14 818

```
24.5.1.3.15 operator!=
                                                                                 [reverse.iter.op!=]
template <class Iterator1, class Iterator2>
  bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
     Returns: x.current != y.current.
24.5.1.3.16 operator>
                                                                                  [reverse.iter.op>]
template <class Iterator1, class Iterator2>
  bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
     Returns: x.current < y.current.
24.5.1.3.17 operator>=
                                                                                [reverse.iter.op>=]
template <class Iterator1, class Iterator2>
  bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
     Returns: x.current <= y.current.
24.5.1.3.18 operator<=
                                                                                [reverse.iter.op <=]
template <class Iterator1, class Iterator2>
  bool operator<=(</pre>
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
     Returns: x.current >= y.current.
24.5.1.3.19 operator-
                                                                                [reverse.iter.opdiff]
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+
                                                                                [reverse.iter.opsum]
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
                                                                                  [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++;
```

§ 24.5.2

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
```

1

[back.insert.iterator]

```
namespace std {
      template <class Container>
       class back_insert_iterator :
         public iterator<output_iterator_tag,void,void,void,void> {
       protected:
         Container* container;
       public:
         typedef Container container_type;
         explicit back_insert_iterator(Container& x);
         back_insert_iterator<Container>&
           operator=(const typename Container::value_type& value);
         back_insert_iterator<Container>&
           operator=(typename Container::value_type&& value);
         back_insert_iterator<Container>& operator*();
         back_insert_iterator<Container>& operator++();
         back_insert_iterator<Container> operator++(int);
       };
      template <class Container>
         back_insert_iterator<Container> back_inserter(Container& x);
   24.5.2.2 back insert iterator operations
                                                                                  [back.insert.iter.ops]
   24.5.2.2.1 back_insert_iterator constructor
                                                                                [back.insert.iter.cons]
   explicit back_insert_iterator(Container& x);
1
        Effects: Initializes container with &x.
                                                                                [back.insert.iter.op=]
   24.5.2.2. back_insert_iterator::operator=
   back_insert_iterator<Container>&
     operator=(const typename Container::value_type& value);
        Effects: container->push_back(value);
2
        Returns: *this.
   back_insert_iterator<Container>&
     operator=(typename Container::value_type&& value);
   § 24.5.2.2.2
                                                                                                    820
```

```
Effects: container->push_back(std::move(value));
4
        Returns: *this.
                                                                                [back.insert.iter.op*]
   24.5.2.2.3 back_insert_iterator::operator*
   back_insert_iterator<Container>& operator*();
        Returns: *this.
   24.5.2.2.4 back_insert_iterator::operator++
                                                                              [back.insert.iter.op++]
   back_insert_iterator<Container>& operator++();
   back_insert_iterator<Container> operator++(int);
        Returns: *this.
   24.5.2.2.5
                back inserter
                                                                                       [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
                                                                                [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=(const typename Container::value_type& 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);
                                                                                [front.insert.iter.ops]
   24.5.2.4 front_insert_iterator operations
   24.5.2.4.1 front_insert_iterator constructor
                                                                               [front.insert.iter.cons]
   explicit front_insert_iterator(Container& x);
        Effects: Initializes container with &x.
```

§ 24.5.2.4.1 821

```
24.5.2.4.2 front_insert_iterator::operator=
                                                                               [front.insert.iter.op=]
   front_insert_iterator<Container>&
     operator=(const typename Container::value_type& value);
1
        Effects: container->push_front(value);
2
        Returns: *this.
   front_insert_iterator<Container>&
     operator=(typename Container::value_type&& value);
        Effects: container->push_front(std::move(value));
        Returns: *this.
                                                                                [front.insert.iter.op*]
   24.5.2.4.3 front_insert_iterator::operator*
   front_insert_iterator<Container>& operator*();
        Returns: *this.
   24.5.2.4.4 front_insert_iterator::operator++
                                                                             [front.insert.iter.op++]
   front_insert_iterator<Container>& operator++();
   front_insert_iterator<Container> operator++(int);
        Returns: *this.
   24.5.2.4.5 front_inserter
                                                                                      [front.inserter]
   template <class Container>
    front_insert_iterator<Container> front_inserter(Container& x);
        Returns: front_insert_iterator<Container>(x).
   24.5.2.5 Class template insert_iterator
                                                                                      [insert.iterator]
    namespace std {
      template <class Container>
       class insert_iterator :
        public iterator<output_iterator_tag,void,void,void,void> {
       protected:
        Container* container;
         typename Container::iterator iter;
      public:
        typedef Container container_type;
         insert_iterator(Container& x, typename Container::iterator i);
         insert_iterator<Container>&
          operator=(const typename Container::value_type& value);
        insert_iterator<Container>&
          operator=(typename Container::value_type&& value);
        insert_iterator<Container>& operator*();
         insert_iterator<Container>& operator++();
         insert_iterator<Container>& operator++(int);
       };
```

§ 24.5.2.5

```
template <class Container>
         insert_iterator<Container> inserter(Container& x, typename Container::iterator i);
   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);
        Effects: Initializes container with &x and iter with i.
   24.5.2.6.2 insert_iterator::operator=
                                                                                      [insert.iter.op=]
   insert_iterator<Container>&
     operator=(const typename Container::value_type& value);
        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;
        Returns: *this.
   24.5.2.6.3 insert iterator::operator*
                                                                                      [insert.iter.op*]
   insert_iterator<Container>& operator*();
        Returns: *this.
                                                                                    [insert.iter.op++]
   24.5.2.6.4 insert_iterator::operator++
   insert_iterator<Container>& operator++();
   insert_iterator<Container>& operator++(int);
        Returns: *this.
   24.5.2.6.5 inserter
                                                                                             [inserter]
   template <class Container>
     insert_iterator<Container> inserter(Container& x, typename Container::iterator i);
        Returns: insert_iterator<Container>(x, i).
                                                                                    [move.iterators]
   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:

§ 24.5.3 823

```
list<string> s;
  // populate the list s
                                                  // copies strings into v1
 vector<string> v1(s.begin(), s.end());
 vector<string> v2(make_move_iterator(s.begin()),
                    make_move_iterator(s.end())); // moves strings into v2
   — end example
24.5.3.1 Class template move_iterator
                                                                                     [move.iterator]
 namespace std {
    template <class Iterator>
    class move_iterator {
   public:
      typedef Iterator
                                                                     iterator_type;
      typedef typename iterator_traits<Iterator>::difference_type
                                                                     difference_type;
      typedef Iterator
                                                                     pointer;
      typedef typename iterator_traits<Iterator>::value_type
                                                                     value_type;
      typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
      typedef 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:
                          // exposition only
      Iterator current;
    };
    template <class Iterator1, class Iterator2>
      bool operator==(
        const move_iterator<!terator1>& x, const move_iterator<!terator2>& 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<=(</pre>
        const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

§ 24.5.3.1

```
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, const move_iterator<Iterator>& x);
       template <class Iterator>
         move_iterator<Iterator> make_move_iterator(const Iterator& i);
     }
   24.5.3.2 move iterator requirements
                                                                              [move.iter.requirements]
  The template parameter Iterator shall meet the requirements for an Input Iterator (24.2.3). Additionally,
   if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall
   meet the requirements for a Bidirectional Iterator (24.2.6) or a Random Access Iterator (24.2.7), respectively.
   24.5.3.3 move_iterator operations
                                                                                         [move.iter.ops]
   24.5.3.3.1 move_iterator constructors
                                                                                    [move.iter.op.const]
   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);
        Effects: Constructs a move_iterator, initializing current with i.
   template <class U> move_iterator(const move_iterator<U>& u);
        Effects: Constructs a move_iterator, initializing current with u.base().
        Requires: U shall be convertible to Iterator.
   24.5.3.3.2 move iterator::operator=
                                                                                        [move.iter.op=]
   template <class U> move_iterator& operator=(const move_iterator<U>& u);
1
        Effects: Assigns u.base() to current.
2
        Requires: U shall be convertible to Iterator.
   24.5.3.3.3 move_iterator conversion
                                                                                     [move.iter.op.conv]
   Iterator base() const;
        Returns: current.
```

§ 24.5.3.3.3 825

```
[move.iter.op.star]
   24.5.3.3.4 move_iterator::operator*
   reference operator*() const;
        Returns: std::move(*current).
   24.5.3.3.5 move_iterator::operator->
                                                                                   [move.iter.op.ref]
   pointer operator->() const;
        Returns: current.
                                                                                  [move.iter.op.incr]
   24.5.3.3.6 move_iterator::operator++
  move_iterator& operator++();
1
        Effects: ++current.
2
        Returns: *this.
  move_iterator operator++(int);
3
        Effects:
          move_iterator tmp = *this;
          ++current;
          return tmp;
   24.5.3.3.7 move_iterator::operator--
                                                                                  [move.iter.op.decr]
   move_iterator& operator--();
        Effects: --current.
2
        Returns: *this.
  move_iterator operator--(int);
3
        Effects:
          move_iterator tmp = *this;
          --current;
          return tmp;
                                                                                     [move.iter.op.+]
   24.5.3.3.8 move_iterator::operator+
  move_iterator operator+(difference_type n) const;
        Returns: move_iterator(current + n).
   24.5.3.3.9 move_iterator::operator+=
                                                                                   [move.iter.op.+=]
  move_iterator& operator+=(difference_type n);
1
        Effects: current += n.
2
        Returns: *this.
   24.5.3.3.10 move_iterator::operator-
                                                                                     [move.iter.op.-]
  move_iterator operator-(difference_type n) const;
1
        Returns: move_iterator(current - n).
   § 24.5.3.3.10
                                                                                                  826
```

```
24.5.3.3.11 move_iterator::operator=
                                                                                    [move.iter.op.-=]
   move_iterator& operator==(difference_type n);
1
        Effects: current -= n.
2
        Returns: *this.
   24.5.3.3.12 move_iterator::operator[]
                                                                                 [move.iter.op.index]
   unspecified operator[](difference_type n) const;
        Returns: std::move(current[n]).
   24.5.3.3.13 move_iterator comparisons
                                                                                 [move.iter.op.comp]
   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<!terator1>& x, const move_iterator<!terator2>& y);
3
        Returns: x.base() < y.base().
   template <class Iterator1, class Iterator2>
   bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
        Returns: !(y < x).
   template <class Iterator1, class Iterator2>
   bool operator>(const move_iterator<!terator1>& x, const move_iterator<!terator2>& y);
5
        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
                                                                             [move.iter.nonmember]
   template <class Iterator1, class Iterator2>
      auto operator-(
      const move_iterator<Iterator1>& x,
      const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
        Returns: x.base() - y.base().
```

§ 24.5.3.3.14 827

```
template <class Iterator>
    move_iterator<Iterator> operator+(
        typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);

2     Returns: x + n.

template <class Iterator>
    move_iterator<Iterator> make_move_iterator(const Iterator& i);

3     Returns: move_iterator<Iterator>(i).
```

### 24.6 Stream iterators

[stream.iterators]

To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

```
[Example:
```

```
partial_sum_copy(istream_iterator<double, char>(cin),
  istream_iterator<double, char>(),
  ostream_iterator<double, char>(cout, "\n"));
```

reads a file containing floating point numbers from cin, and prints the partial sums onto cout. — end example

## 24.6.1 Class template istream\_iterator

[istream.iterator]

- The class template istream\_iterator is an input iterator (24.2.3) that reads (using operator>>) successive elements from the input stream for which it was constructed. After it is constructed, and every time ++ is used, the iterator reads and stores a value of T. If the iterator fails to read and store a value of T (fail() on the stream returns true), the iterator becomes equal to the end-of-stream iterator value. The constructor with no arguments istream\_iterator() always constructs an end-of-stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator\* on an end-of-stream iterator is not defined. For any other iterator value a const T& is returned. The result of operator-> on an end-of-stream iterator is not defined. For any other iterator value a const T\* is returned. The behavior of a program that applies operator++() to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators.
- Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```
namespace std {
 template <class T, class charT = char, class traits = char_traits<charT>,
      class Distance = ptrdiff_t>
 class istream_iterator:
   public 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++();
```

§ 24.6.1

```
istream_iterator<T,charT,traits,Distance> operator++(int);
       private:
         basic_istream<charT,traits>* in_stream; // exposition only
                                                   // exposition only
         T value;
       };
       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);
     }
                                                                                  [istream.iterator.cons]
   24.6.1.1 istream_iterator constructors and destructor
   see below istream_iterator();
1
        Effects: Constructs the end-of-stream iterator. If T is a literal type, then this constructor shall be a
        constexpr constructor.
2
        Postcondition: in_stream == 0.
   istream_iterator(istream_type& s);
3
        Effects: Initializes in_stream with &s. value may be initialized during construction or the first time it
        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.
                                                                                   [istream.iterator.ops]
   24.6.1.2 istream_iterator operations
   const T& operator*() const;
        Returns: value.
   const T* operator->() const;
        Returns: &(operator*()).
   istream_iterator<T,charT,traits,Distance>& operator++();
3
         Requires: in_stream != 0.
4
        Effects: *in_stream > value.
5
        Returns: *this.
   § 24.6.1.2
                                                                                                       829
```

```
istream_iterator<T,charT,traits,Distance> operator++(int);
6
        Requires: in_stream != 0.
        Effects:
          istream_iterator<T,charT,traits,Distance> tmp = *this;
          *in_stream >> value;
          return (tmp);
   template <class T, class charT, class traits, class Distance>
     bool operator==(const istream_iterator<T,charT,traits,Distance> &x,
                     const istream_iterator<T,charT,traits,Distance> &y);
8
        Returns: x.in_stream == y.in_stream.
   template <class T, class charT, class traits, class Distance>
     bool operator!=(const istream_iterator<T,charT,traits,Distance> &x,
                     const istream_iterator<T,charT,traits,Distance> &y);
9
        Returns: !(x == y)
```

## 24.6.2 Class template ostream\_iterator

[ostream.iterator]

ostream\_iterator writes (using operator<<) successive elements onto the output stream from which it was constructed. If it was constructed with charT\* as a constructor argument, this string, called a *delimiter string*, is written to the stream after every T is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```
while (first != last)
    *result++ = *first++;
ostream_iterator is defined as:
  namespace std {
    template <class T, class charT = char, class traits = char_traits<charT> >
    class ostream_iterator:
      public iterator<output_iterator_tag, void, void, void, void> {
    public:
      typedef charT char_type;
      typedef traits traits_type;
      typedef basic_ostream<charT,traits> ostream_type;
      ostream_iterator(ostream_type& s);
      ostream_iterator(ostream_type& s, const charT* delimiter);
      ostream_iterator(const ostream_iterator<T,charT,traits>& x);
     ~ostream_iterator();
      ostream_iterator<T,charT,traits>& operator=(const T& value);
      ostream_iterator<T,charT,traits>& operator*();
      ostream_iterator<T,charT,traits>& operator++();
      ostream_iterator<T,charT,traits>& operator++(int);
      basic_ostream<charT,traits>* out_stream; // exposition only
      const charT* delim;
                                                 // exposition only
    };
  }
```

§ 24.6.2

```
ostream_iterator constructors and destructor
                                                                             [ostream.iterator.cons.des]
   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();
        Effects: The iterator is destroyed.
             ostream_iterator operations
                                                                                  [ostream.iterator.ops]
   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

[istreambuf.iterator]

The class template <code>istreambuf\_iterator</code> defines an input iterator (24.2.3) that reads successive characters from the streambuf for which it was constructed. <code>operator\*</code> 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 <code>traits::eof()</code>), the iterator becomes equal to the end-of-stream iterator value. The default constructor <code>istreambuf\_iterator()</code> and the constructor <code>istreambuf\_iterator()</code> both construct an end-of-stream iterator object suitable for use as an end-of-range. All specializations of <code>istreambuf\_iterator</code> shall have a trivial copy constructor, a <code>constexpr</code> 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.

§ 24.6.3

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;
                                            traits_type;
      typedef traits
      typedef typename traits::int_type
                                            int_type;
      typedef basic_streambuf<charT,traits> streambuf_type;
      typedef basic_istream<charT,traits>
                                            istream_type;
      class proxy;
                                             // exposition only
      constexpr istreambuf_iterator() noexcept;
      istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
      ~istreambuf_iterator() = default;
      istreambuf_iterator(istream_type& s) noexcept;
      istreambuf_iterator(streambuf_type* s) noexcept;
      istreambuf_iterator(const proxy& p) noexcept;
      charT operator*() const;
      pointer operator->() const;
      istreambuf_iterator<charT,traits>& operator++();
      proxy operator++(int);
      bool equal(const istreambuf_iterator& b) const;
    private:
                                             // exposition only
      streambuf_type* sbuf_;
    };
    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
                                                                        [istreambuf.iterator::proxy]
 namespace std {
    template <class charT, class traits = char_traits<charT> >
    class istreambuf_iterator<charT, traits>::proxy {
      charT keep_;
      basic_streambuf<charT,traits>* sbuf_;
      proxy(charT c,
       basic_streambuf<charT,traits>* sbuf)
        : keep_(c), sbuf_(sbuf) { }
   public:
      charT operator*() { return keep_; }
 }
```

Class istreambuf\_iterator<charT, traits>::proxy is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class istreambuf\_-iterator<charT, traits>::proxy provides a temporary placeholder as the return value of the post-

§ 24.6.3.1

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
                                                                             [istreambuf.iterator.cons]
   constexpr istreambuf_iterator() noexcept;
1
        Effects: Constructs the end-of-stream iterator.
   istreambuf_iterator(basic_istream<charT,traits>& s) noexcept;
   istreambuf_iterator(basic_streambuf<charT,traits>* s) noexcept;
        Effects: Constructs an istreambuf_iterator<> that uses the basic_streambuf<> object *(s.rdbuf()),
        or *s, respectively. Constructs an end-of-stream iterator if s.rdbuf() is null.
   istreambuf_iterator(const proxy& p) noexcept;
3
        Effects: Constructs a istreambuf_iterator<> that uses the basic_streambuf<> object pointed to
        by the proxy object's constructor argument p.
   24.6.3.3 istreambuf_iterator::operator*
                                                                             [istreambuf.iterator::op*]
   charT operator*() const
        Returns: The character obtained via the streambuf member sbuf_->sgetc().
   24.6.3.4
             istreambuf_iterator::operator++
                                                                          [istreambuf.iterator::op++]
   istreambuf_iterator<charT,traits>&
       istreambuf_iterator<charT,traits>::operator++();
1
        Effects: sbuf_->sbumpc().
        Returns: *this.
   proxy istreambuf_iterator<charT,traits>::operator++(int);
3
        Returns: proxy(sbuf_-->sbumpc(), sbuf_).
   24.6.3.5 istreambuf_iterator::equal
                                                                           [istreambuf.iterator::equal]
   bool equal(const istreambuf_iterator<charT,traits>& b) const;
1
        Returns: true if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regard-
        less of what streambuf object they use.
   24.6.3.6
            operator==
                                                                          [istreambuf.iterator::op==]
   template <class charT, class traits>
     bool operator==(const istreambuf_iterator<charT,traits>& a,
                     const istreambuf_iterator<charT,traits>& b);
        Returns: a.equal(b).
   24.6.3.7 operator!=
                                                                           [istreambuf.iterator::op!=]
   template <class charT, class traits>
     bool operator!=(const istreambuf_iterator<charT,traits>& a,
                     const istreambuf_iterator<charT,traits>& b);
        Returns: !a.equal(b).
   § 24.6.3.7
                                                                                                     833
```

## 24.6.4 Class template ostreambuf\_iterator

[ostreambuf.iterator]

```
namespace std {
       template <class charT, class traits = char_traits<charT> >
       class ostreambuf_iterator :
         public iterator<output_iterator_tag, void, void, void, void> {
       public:
         typedef charT
                                                char_type;
         typedef traits
                                                traits_type;
         typedef basic_streambuf<charT,traits> streambuf_type;
         typedef basic_ostream<charT,traits>
                                                ostream_type;
       public:
         ostreambuf_iterator(ostream_type& s) noexcept;
         ostreambuf_iterator(streambuf_type* s) noexcept;
         ostreambuf_iterator& operator=(charT c);
         ostreambuf_iterator& operator*();
         ostreambuf_iterator& operator++();
         ostreambuf_iterator& operator++(int);
         bool failed() const noexcept;
       private:
                                                // exposition only
         streambuf_type* sbuf_;
       };
  The class template ostreambuf_iterator writes successive characters onto the output stream from which
   it was constructed. It is not possible to get a character value out of the output iterator.
            ostreambuf_iterator constructors
                                                                                 [ostreambuf.iter.cons]
   ostreambuf_iterator(ostream_type& s) noexcept;
        Requires: s.rdbuf() shall not null pointer.
2
        Effects: :sbuf_(s.rdbuf()) {}.
   ostreambuf_iterator(streambuf_type* s) noexcept;
3
        Requires: s shall not be a null pointer.
        Effects: sbuf_(s) {}.
   24.6.4.2 ostreambuf_iterator operations
                                                                                   [ostreambuf.iter.ops]
   ostreambuf_iterator<charT,traits>&
     operator=(charT c);
        Effects: If failed() yields false, calls sbuf_->sputc(c); otherwise has no effect.
        Returns: *this.
   ostreambuf_iterator<charT,traits>& operator*();
3
        Returns: *this.
```

§ 24.6.4.2 834

1

```
ostreambuf_iterator<charT,traits>& operator++();
   ostreambuf_iterator<charT,traits>& operator++(int);
        Returns: *this.
   bool failed() const noexcept;
        Returns: true if in any prior use of member operator=, the call to sbuf_->sputc() returned
        traits::eof(); or false otherwise.
                                                                                      [iterator.range]
   24.6.5
           range access
<sup>1</sup> 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>, st>,
   <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());
2
        Returns: c.begin().
   template <class C> auto end(C& c) -> decltype(c.end());
   template <class C> auto end(const C& c) -> decltype(c.end());
3
        Returns: c.end().
   template <class T, size_t N> T* begin(T (&array)[N]);
4
        Returns: array.
   template <class T, size_t N> T* end(T (&array)[N]);
        Returns: array + N.
```

§ 24.6.5

# 25 Algorithms library

# [algorithms]

### 25.1 General

[algorithms.general]

This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause 23) and other sequences.

The following subclauses describe components for non-modifying sequence operation, modifying sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 112.

Table 112 — Algorithms library summary

	Subclause	Header(s)
25.2	Non-modifying sequence operations	
25.3	Mutating sequence operations	<algorithm></algorithm>
25.4	Sorting and related operations	
25.5	C library algorithms	<cstdlib></cstdlib>

## Header <algorithm> synopsis

```
#include <initializer_list>
namespace std {
  // 25.2, non-modifying sequence operations:
  template <class InputIterator, class Predicate>
    bool all_of(InputIterator first, InputIterator last, Predicate pred);
  template <class InputIterator, class Predicate>
    bool any_of(InputIterator first, InputIterator last, Predicate pred);
 template <class InputIterator, class Predicate>
    bool none_of(InputIterator first, InputIterator last, Predicate pred);
 template < class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function f);
  template < class InputIterator, class T>
    InputIterator find(InputIterator first, InputIterator last,
                       const T& value);
  template < class InputIterator, class Predicate >
    InputIterator find_if(InputIterator first, InputIterator last,
                          Predicate pred);
  template < class InputIterator, class Predicate >
    InputIterator find_if_not(InputIterator first, InputIterator last,
                               Predicate pred);
  template<class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1
      find_end(ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2);
  template < class Forward Iterator 1, class Forward Iterator 2,
     class BinaryPredicate>
    ForwardIterator1
```

```
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
template < class InputIterator, class ForwardIterator>
  InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2);
template<class InputIterator, class ForwardIterator,
   class BinaryPredicate>
  InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate pred);
template<class ForwardIterator>
 ForwardIterator adjacent_find(ForwardIterator first,
                                ForwardIterator last);
template < class Forward Iterator, class Binary Predicate >
 ForwardIterator adjacent_find(ForwardIterator first,
                                ForwardIterator last,
                                BinaryPredicate pred);
template<class InputIterator, class T>
  typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last, const T& value);
template < class InputIterator, class Predicate >
  typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred);
template<class InputIterator1, class InputIterator2>
 pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template
 <class InputIterator1, class InputIterator2, class BinaryPredicate>
 pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
template<class InputIterator1, class InputIterator2>
 bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template
 <class InputIterator1, class InputIterator2, class BinaryPredicate>
 bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2>
 bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2);
template < class Forward Iterator 1, class Forward Iterator 2,
                 class BinaryPredicate>
 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 Forward Iterator 1, class Forward Iterator 2,
   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>
 ForwardIterator1 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 Forward Iterator, 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 Forward Iterator, class Generator >
  void generate (ForwardIterator first, ForwardIterator last,
                Generator gen);
template<class OutputIterator, class Size, class Generator>
  OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
template<class ForwardIterator, class T>
  ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                         const T& value);
template<class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred);
template<class InputIterator, class OutputIterator, class T>
  OutputIterator remove_copy(InputIterator first, InputIterator last,
                             OutputIterator result, const T& value);
template < class InputIterator, class OutputIterator, class Predicate >
  OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                                OutputIterator result, Predicate pred);
template<class ForwardIterator>
  ForwardIterator unique(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);
template<class InputIterator, class OutputIterator>
  OutputIterator unique_copy(InputIterator first, InputIterator last,
                             OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryPredicate>
  OutputIterator unique_copy(InputIterator first, InputIterator last,
```

```
OutputIterator result, BinaryPredicate pred);
template < class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
template<class BidirectionalIterator, class OutputIterator>
  OutputIterator reverse_copy(BidirectionalIterator first,
                               BidirectionalIterator last,
                               OutputIterator result);
template < class ForwardIterator >
  ForwardIterator rotate(ForwardIterator first, ForwardIterator middle,
                          ForwardIterator last);
template < class ForwardIterator, class OutputIterator>
  OutputIterator rotate_copy(
    ForwardIterator first, ForwardIterator middle,
    ForwardIterator last, OutputIterator result);
template < class Random Access Iterator >
  void random_shuffle(RandomAccessIterator first,
                       RandomAccessIterator last);
template < class Random Access Iterator, class Random Number Generator >
  void random_shuffle(RandomAccessIterator first,
                       RandomAccessIterator last,
                       RandomNumberGenerator&& rand);
template < class Random AccessIterator, class Uniform Random Number Generator >
  void shuffle(RandomAccessIterator first,
                       RandomAccessIterator last,
                       UniformRandomNumberGenerator&& rand);
// 25.3.13, partitions:
template <class InputIterator, class Predicate>
  bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);
template < class Forward Iterator, class Predicate >
  ForwardIterator partition(ForwardIterator first,
                             ForwardIterator last,
                             Predicate pred);
template < class BidirectionalIterator, class Predicate >
  BidirectionalIterator stable_partition(BidirectionalIterator first,
                                          BidirectionalIterator last,
                                          Predicate pred);
template <class InputIterator, class OutputIterator1,
          class OutputIterator2, class Predicate>
  pair<OutputIterator1, OutputIterator2>
  partition_copy(InputIterator first, InputIterator last,
                 OutputIterator1 out_true, OutputIterator2 out_false,
                 Predicate pred);
template < class Forward Iterator, class Predicate >
  ForwardIterator partition_point(ForwardIterator first,
                                   ForwardIterator last,
                                   Predicate pred);
// 25.4, sorting and related operations:
// 25.4.1, sorting:
template<class RandomAccessIterator>
```

```
void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
template < class Random Access Iterator >
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
template < class Random Access Iterator, 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 Random Access Iterator, 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 Forward Iterator, class Compare >
  bool is_sorted(ForwardIterator first, ForwardIterator last,
                 Compare comp);
template<class ForwardIterator>
  ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);
template < class Forward Iterator, 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 Random AccessIterator, class Compare >
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
```

```
Compare comp);
template<class RandomAccessIterator>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last);
template < class Random AccessIterator, class Compare >
  void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
template < class Random Access Iterator, class Compare >
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
template<class RandomAccessIterator>
  bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
template < class Random Access Iterator, class Compare >
  bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
template<class RandomAccessIterator>
  RandomAccessIterator is heap until(RandomAccessIterator first, RandomAccessIterator last);
template < class Random Access Iterator, class Compare >
  RandomAccessIterator is_heap_until(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, class Compare >
  const T& min(const T& a, const T& b, Compare comp);
template<class T>
  T min(initializer_list<T> t);
template < class T, class Compare >
  T min(initializer_list<T> t, Compare comp);
template<class T> const T& max(const T& a, const T& b);
template < class T, class Compare >
  const T& max(const T& a, const T& b, Compare comp);
template<class T>
  T max(initializer_list<T> t);
template < class T, class Compare >
  T max(initializer_list<T> t, Compare comp);
template<class T> pair<const T&, const T&> minmax(const T& a, const T& b);
template < class T, class Compare >
  pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
template<class T>
  pair<T, T> minmax(initializer_list<T> t);
template < class T, class Compare >
  pair<T, T> minmax(initializer_list<T> t, Compare comp);
template<class ForwardIterator>
  ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
template < class Forward Iterator, class Compare >
  ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                               Compare comp);
template<class ForwardIterator>
```

```
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
  template < class Forward Iterator, class Compare >
    ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                 Compare comp);
  template < class ForwardIterator >
    pair<ForwardIterator, ForwardIterator>
      minmax_element(ForwardIterator first, ForwardIterator last);
  template < class Forward Iterator, 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);
}
```

- <sup>3</sup> 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.
- <sup>4</sup> 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.
- 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, 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, or RandomAccessIterator2, the actual template argument shall satisfy the requirements of a random-access iterator (24.2.7).
- <sup>6</sup> 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]

- <sup>7</sup> Both in-place and copying versions are provided for certain algorithms.<sup>268</sup> When such a version is provided for *algorithm* it is called *algorithm\_copy*. Algorithms that take predicates end with the suffix \_if (which follows the suffix \_copy).
- The Predicate parameter is used whenever an algorithm expects a function object (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 (Clause 4). The function object pred shall not apply any non-constant function through the dereferenced iterator.
- The BinaryPredicate parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type T when T is part of the signature returns a value testable as true. In other words, if an algorithm takes BinaryPredicate binary\_pred as its argument and first1 and first2 as its iterator arguments, it should work correctly in the construct binary\_pred(\*first1, \*first2) contextually converted to bool (Clause 4). BinaryPredicate always takes the first iterator's value\_type as its first argument, that is, in those cases when T value is part of the signature, it should work correctly in the construct binary\_pred(\*first1, value) contextually converted to bool (Clause 4). binary\_pred shall not apply any non-constant function through the dereferenced iterators.
- [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> (20.8.3), or some equivalent solution. end note]
- When the description of an algorithm gives an expression such as \*first == value for a condition, the expression shall evaluate to either true or false in boolean contexts.
- 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

```
X tmp = a;
advance(tmp, n);
return tmp;
and that of b-a is the same as of
return distance(a, b);
```

## 25.2 Non-modifying sequence operations

[alg.nonmodifying]

#### 25.2.1 All of

1

2

[alg.all of]

```
template <class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last, Predicate pred);
```

Returns: true if [first,last) is empty or if pred(\*i) is true for every iterator i in the range [first,last), and false otherwise.

Complexity: At most last - first applications of the predicate.

25.2.2 Any of [alg.any\_of]

§ 25.2.2 846

<sup>268)</sup> 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.

```
template <class InputIterator, class Predicate>
     bool any_of(InputIterator first, InputIterator last, Predicate pred);
        Returns: false if [first,last) is empty or if there is no iterator i in the range [first,last) such
        that pred(*i) is true, and true otherwise.
2
        Complexity: At most last - first applications of the predicate.
   25.2.3
           None of
                                                                                          [alg.none_of]
   template <class InputIterator, class Predicate>
     bool none_of(InputIterator first, InputIterator last, Predicate pred);
1
        Returns: true if [first,last) is empty or if pred(*i) is false for every iterator i in the range
        [first, last), and false otherwise.
        Complexity: At most last - first applications of the predicate.
                                                                                           [alg.foreach]
   25.2.4 For each
   template < class InputIterator, class Function>
     Function for_each(InputIterator first, InputIterator last, Function f);
1
        Requires: Function shall meet the requirements of MoveConstructible (Table 20). [Note: Function
        need not meet the requirements of CopyConstructible (Table 21). — end note
2
        Effects: Applies f to the result of dereferencing every iterator in the range [first,last), starting
        from first and proceeding to last - 1. [Note: If the type of first satisfies the requirements of a
        mutable iterator, f may apply nonconstant functions through the dereferenced iterator. — end note]
3
        Returns: std::move(f).
4
        Complexity: Applies f exactly last - first times.
5
        Remarks: If f returns a result, the result is ignored.
   25.2.5 Find
                                                                                               [alg.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);
1
        Returns: The first iterator i in the range [first,last) for which the following corresponding condi-
        tions hold: *i == value, pred(*i) != false, pred(*i) == false. Returns last if no such iterator
        is found.
2
        Complexity: At most last - first applications of the corresponding predicate.
   25.2.6 Find end
                                                                                          [alg.find.end]
   template < class Forward Iterator 1, class Forward Iterator 2>
     ForwardIterator1
       find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2);
```

§ 25.2.6 847

1

1

```
template < class Forward Iterator 1, class Forward Iterator 2,
            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.
2
        Returns: The last iterator i in the range [first1,last1 - (last2 - first2)) such that for any non-
        negative integer n < (last2 - first2), the following corresponding conditions hold: *(i + n) ==
        *(first2 + n), pred(*(i + n), *(first2 + n)) != false. Returns last1 if [first2,last2) is
        empty or if no such iterator is found.
3
        Complexity: At most (last2 - first2) * (last1 - first1 - (last2 - first2) + 1) applica-
        tions 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>
     {\tt 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.
2
        Returns: The first iterator i in the range [first1,last1) such that for some iterator j in the range
        [first2,last2) the following conditions hold: *i == *j, pred(*i,*j) != false. Returns last1
        if [first2,last2) is empty or if no such iterator is found.
3
        Complexity: At most (last1-first1) * (last2-first2) applications of the corresponding predi-
        cate.
                                                                                    [alg.adjacent.find]
   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) applica-
        tions of the corresponding predicate, where i is adjacent_find's return value.
   25.2.9
                                                                                             [alg.count]
           Count
```

§ 25.2.9 848

1

1

2

2

```
template < class InputIterator, class T>
    typename iterator_traits<InputIterator>::difference_type
       count(InputIterator first, InputIterator last, const T& value);
template < class InputIterator, class Predicate >
    typename iterator_traits<InputIterator>::difference_type
      count_if(InputIterator first, InputIterator last, Predicate pred);
     Effects: Returns the number of iterators i in the range [first,last) for which the following corre-
     sponding conditions hold: *i == value, pred(*i) != false.
     Complexity: Exactly last - first applications of the corresponding predicate.
                                                                                        [mismatch]
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);
     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.
     Complexity: At most last1 - first1 applications of the corresponding predicate.
                                                                                         [alg.equal]
25.2.11 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);
     Returns: true if for every iterator i in the range [first1,last1) the following corresponding condi-
     tions hold: *i == *(first2 + (i - first1)), pred(*i, *(first2 + (i - first1))) != false.
     Otherwise, returns false.
     Complexity: At most last1 - first1 applications of the corresponding predicate.
         Is permutation
                                                                             [alg.is_permutation]
25.2.12
template<class ForwardIterator1, class ForwardIterator2>
 bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2);
template < class Forward Iterator 1, class Forward Iterator 2,
§ 25.2.12
                                                                                                  849
```

1

2

3

1

2

3

4

5

6

7

```
class BinaryPredicate>
 bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, BinaryPredicate pred);
     Requires:: ForwardIterator1 and ForwardIterator2 shall have the same value type. The comparison
     function shall be an equivalence relation.
     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.
     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 re-
     turn true; otherwise, at worst \mathcal{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 Forward Iterator 1, class Forward Iterator 2,
         class BinaryPredicate>
 {\tt ForwardIterator1}
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
     Effects: Finds a subsequence of equal values in a sequence.
     Returns: The first iterator i in the range [first1,last1 - (last2-first2)) such that for any non-
     \label{eq: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, otherwise returns last1 if no such iterator is found.
     Complexity: At most (last1 - first1) * (last2 - first2) applications of the corresponding
     predicate.
template < class Forward Iterator, class Size, class T>
 ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last, Size count,
           const T& value);
template < class Forward Iterator, 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 +
```

§ 25.2.13 850

Complexity: At most last - first applications of the corresponding predicate.

n), value) != false. Returns last if no such iterator is found.

[alg.modifying.operations]

Mutating sequence operations

```
[alg.copy]
   25.3.1 Copy
   template < class InputIterator, class OutputIterator>
      OutputIterator copy(InputIterator first, InputIterator last,
                          OutputIterator result);
1
         Effects: Copies elements in the range [first,last) into the range [result,result + (last -
         first)) starting from first and proceeding to last. For each non-negative integer n < (last -
         first), performs *(result + n) = *(first + n).
2
         Returns: result + (last - first).
3
         Requires: result shall not be in the range [first,last).
4
         Complexity: Exactly last - first assignments.
   template < class InputIterator, class Size, class OutputIterator>
      OutputIterator copy_n(InputIterator first, Size n,
                            OutputIterator result);
5
         Effects: For each non-negative integer i < n, performs *(result + i) = *(first + i).
6
         Returns: result + n.
7
         Complexity: Exactly n assignments.
   template < class InputIterator, class OutputIterator, class Predicate >
      OutputIterator copy_if(InputIterator first, InputIterator last,
                             OutputIterator result, Predicate pred);
8
         Requires: The ranges [first,last) and [result,result + (last - first)) shall not overlap.
9
         Effects: Copies all of the elements referred to by the iterator i in the range [first,last) for which
         pred(*i) is true.
10
         Complexity: Exactly last - first applications of the corresponding predicate.
11
         Remarks: Stable.
   template < class BidirectionalIterator1, class BidirectionalIterator2>
      BidirectionalIterator2
        copy_backward(BidirectionalIterator1 first,
                      BidirectionalIterator1 last,
                      BidirectionalIterator2 result);
12
         Effects: Copies elements in the range [first,last) into the range [result - (last-first),result
         ) starting from last - 1 and proceeding to first. <sup>269</sup> For each positive integer n <= (last - first),
         performs *(result - n) = *(last - n).
13
         Requires: result shall not be in the range (first, last].
14
         Returns: result - (last - first).
15
         Complexity: Exactly last - first assignments.
   269) copy_backward should be used instead of copy when last is in the range [result - (last - first),result).
```

§ 25.3.1 851

```
25.3.2 Move
                                                                                             [alg.move]
   template < class InputIterator, class OutputIterator>
     OutputIterator move(InputIterator first, InputIterator last,
                         OutputIterator result);
        Effects: Moves elements in the range [first,last) into the range [result,result + (last -
        first)) starting from first and proceeding to last. For each non-negative integer n < (last-first),
        performs *(result + n) = std::move(*(first + n)).
        Returns: result + (last - first).
3
        Requires: result shall not be in the range [first,last).
4
        Complexity: Exactly last - first move assignments.
   template<class BidirectionalIterator1, class BidirectionalIterator2>
     BidirectionalIterator2
       move_backward(BidirectionalIterator1 first,
                     BidirectionalIterator1 last,
                     BidirectionalIterator2 result);
5
        Effects: Moves elements in the range [first,last) into the range [result - (last-first),result
        ) starting from last - 1 and proceeding to first. <sup>270</sup> For each positive integer n <= (last - first),
        performs *(result - n) = std::move(*(last - n)).
6
        Requires: result shall not be in the range (first, last].
7
        Returns: result - (last - first).
8
        Complexity: Exactly last - first assignments.
                                                                                             [alg.swap]
   25.3.3 swap
   template<class ForwardIterator1, class ForwardIterator2>
     ForwardIterator2
       swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2);
1
        Effects: For each non-negative integer n < (last1 - first1) performs: swap(*(first1 + n),
        Requires: The two ranges [first1,last1) and [first2,first2 + (last1 - first1)) shall not
        overlap. *(first1 + n) shall be swappable with (17.6.3.2) *(first2 + n).
3
        Returns: first2 + (last1 - first1).
4
        Complexity: Exactly last1 - first1 swaps.
   template < class ForwardIterator1, class ForwardIterator2>
     void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
5
        Effects: swap(*a, *b).
6
        Requires: a and b shall be dereferenceable. *a shall be swappable with (17.6.3.2) *b.
   270) move_backward should be used instead of move when last is in the range [result - (last - first), result).
```

§ 25.3.3 852

```
25.3.4 Transform
                                                                                   [alg.transform]
template < class InputIterator, class OutputIterator,
         class UnaryOperation>
  OutputIterator
    transform(InputIterator first, InputIterator last,
              OutputIterator result, UnaryOperation op);
template < class InputIterator1, class InputIterator2,
         class OutputIterator, class BinaryOperation>
  OutputIterator
    transform(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);
     Effects: Assigns through every iterator i in the range [result, result + (last1 - first1)) a
     new corresponding value equal to op(*(first1 + (i - result)) or binary_op(*(first1 + (i -
     result)), *(first2 + (i - result))).
     Requires: op and binary_op shall not invalidate iterators or subranges, or modify elements in the
     ranges [first1,last1], [first2,first2 + (last1 - first1)], and [result,result + (last1 -
     first1)].^{271}
     Returns: result + (last1 - first1).
     Complexity: Exactly last1 - first1 applications of op or binary_op.
     Remarks: result may be equal to first in case of unary transform, or to first1 or first2 in case
     of binary transform.
25.3.5 Replace
                                                                                       [alg.replace]
template < class ForwardIterator, class T>
  void replace(ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);
template < class Forward Iterator, class Predicate, class T>
  void replace_if(ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);
     Requires: The expression *first = new_value shall be valid.
     Effects: Substitutes elements referred by the iterator i in the range [first,last) with new_value,
     when the following corresponding conditions hold: *i == old_value, pred(*i) != false.
     Complexity: Exactly last - first applications of the corresponding predicate.
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);
271) The use of fully closed ranges is intentional.
```

1

2

3

4

5

1

2

3

§ 25.3.5

```
Requires: The results of the expressions *first and new_value shall be writable to the result output iterator. The ranges [first,last) and [result,result + (last - first)) shall not overlap.
```

Effects: Assigns to every iterator i in the range [result,result + (last - first)) either new\_-value or \*(first + (i - result)) depending on whether the following corresponding conditions hold:

```
*(first + (i - result)) == old_value
pred(*(first + (i - result))) != false
```

6 Returns: result + (last - first).

1

7 Complexity: Exactly last - first applications of the corresponding predicate.

25.3.6 Fill [alg.fill]

```
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);
```

- Requires: The expression value shall be writable to the output iterator. The type Size shall be convertible to an integral type (4.7, 12.3).
- 2 Effects: The first algorithm assigns value through all the iterators in the range [first,last). The second algorithm assigns value through all the iterators in the range [first,first + n) if n is positive, otherwise it does nothing.
- 3 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 [alg.generate]

template<class OutputIterator, class Size, class Generator>
 OutputIterator generate\_n(OutputIterator first, Size n, Generator gen);

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 [alg.remove]

§ 25.3.8 854

```
1 Requires: The type of *first shall satisfy the MoveAssignable requirements (Table 22).
```

- 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.
- Note: each element in the range [ret,last), where ret is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by swapping with or moving from elements that were originally in that range.

- Requires: The ranges [first,last) and [result,result + (last first)) shall not overlap. The expression \*result = \*first shall ve valid.
- 8 Effects: Copies all the elements referred to by the iterator i in the range [first,last) for which the following corresponding conditions do not hold: \*i == value, pred(\*i) != false.
- 9 Returns: The end of the resulting range.
- 10 Complexity: Exactly last first applications of the corresponding predicate.
- 11 Remarks: Stable.

25.3.9 Unique [alg.unique]

- Effects: For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator i in the range [first + 1,last) for which the following conditions hold: \*(i 1) == \*i or pred(\*(i 1), \*i) != false.
- Requires: The comparison function shall be an equivalence relation. The type of \*first shall satisfy the MoveAssignable requirements (Table 22).
- 3 Returns: The end of the resulting range.
- 4 Complexity: For nonempty ranges, exactly (last first) 1 applications of the corresponding predicate.

§ 25.3.9 855

5

6

7

8

1

2

3

4

5

6

7

```
template < class InputIterator, class OutputIterator>
  OutputIterator
   unique_copy(InputIterator first, InputIterator last,
                OutputIterator result);
template < class InputIterator, class OutputIterator,
         class BinaryPredicate>
  OutputIterator
   unique_copy(InputIterator first, InputIterator last,
                OutputIterator result, BinaryPredicate pred);
     Requires: The comparison function shall be an equivalence relation. The ranges [first,last) and
     [result,result+(last-first)) shall not overlap. The expression *result = *first shall be valid.
     If neither InputIterator nor OutputIterator meets the requirements of forward iterator then the
     value type of InputIterator shall be CopyConstructible (Table 21) and CopyAssignable (Table 23).
     Otherwise CopyConstructible is not required.
     Effects: Copies only the first element from every consecutive group of equal elements referred to by
     the iterator i in the range [first,last) for which the following corresponding conditions hold: *i
     == *(i - 1) \text{ or pred}(*i, *(i - 1)) != false.
     Returns: The end of the resulting range.
     Complexity: For nonempty ranges, exactly last - first - 1 applications of the corresponding pred-
     icate.
25.3.10 Reverse
                                                                                       [alg.reverse]
template < class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
     Effects: For each non-negative integer i <= (last - first)/2, applies iter_swap to all pairs of
     iterators first + i, (last - i) - 1.
     Requires: *first shall be swappable (17.6.3.2).
     Complexity: Exactly (last - first)/2 swaps.
template < class BidirectionalIterator, class OutputIterator >
  OutputIterator
    reverse_copy(BidirectionalIterator first,
                 BidirectionalIterator last, OutputIterator result);
     Effects: Copies the range [first,last) to the range [result,result+(last-first)) such that
     for any non-negative integer i < (last - first) the following assignment takes place: *(result +
     (last - first) - i) = *(first + i).
     Requires: The ranges [first,last) and [result,result+(last-first)) shall not overlap.
     Returns: result + (last - first).
     Complexity: Exactly last - first assignments.
25.3.11
          Rotate
                                                                                        [alg.rotate]
template < class ForwardIterator>
 ForwardIterator rotate(ForwardIterator first, ForwardIterator middle,
              ForwardIterator last);
```

§ 25.3.11 856

```
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.
        Requires: [first,middle) and [middle,last) shall be valid ranges. ForwardIterator shall satisfy
        the requirements of ValueSwappable (17.6.3.2). The type of *first shall satisfy the requirements of
        MoveConstructible (Table 20) and the requirements of MoveAssignable (Table 22).
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.
                                                                                [alg.random.shuffle]
   25.3.12 Random shuffle
   template < class Random Access Iterator >
     void random_shuffle(RandomAccessIterator first,
                         RandomAccessIterator last);
   template < class Random Access Iterator, class Random Number Generator >
     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 [first,last) such that each possible permutation of
        those elements has equal probability of appearance.
2
        Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). 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
```

- iterator\_traits<RandomAccessIterator>::difference\_type.

  Complexity: Exactly (last 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:

a randomly chosen value in the interval [0,n), for n > 0 of type iterator\_traits<RandomAccess-Iterator>::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

§ 25.3.12 857

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 [alg.partitions]

template <class InputIterator, class Predicate>
bool is\_partitioned(InputIterator first, InputIterator last, Predicate pred);

Requires: InputIterator's value type shall be convertible to Predicate's argument type.

- Returns: true if [first,last) is empty or if [first,last) is partitioned by pred, i.e. if all elements that satisfy pred appear before those that do not.
- 3 Complexity: Linear. At most last first applications of pred.

1

- 4 Effects: Places all the elements in the range [first,last) that satisfy pred before all the elements that do not satisfy it.
- Returns: An iterator i such that for any iterator j in the range [first,i) pred(\*j) != false, and for any iterator k in the range [i,last), pred(\*k) == false.
- 6 Requires: ForwardIterator shall satisfy the requirements of ValueSwappable (17.6.3.2).
- Complexity: If ForwardIterator meets the requirements for a BidirectionalIterator, at most (last first) / 2 swaps are done; otherwise at most last first swaps are done. Exactly last first applications of the predicate are done.

- Effects: Places all the elements in the range [first,last) that satisfy pred before all the elements that do not satisfy it.
- Returns: An iterator i such that for any iterator j in the range [first,i), pred(\*j) != false, and for any iterator k in the range [i,last), pred(\*k) == false. The relative order of the elements in both groups is preserved.
- Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The type of \*first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).
- Complexity: At most (last first) \* log(last first) swaps, but only linear number of swaps if there is enough extra memory. Exactly last first applications of the predicate.

§ 25.3.13 858

Requires: InputIterator's value type shall be Assignable, and shall be writable to the out\_true and out\_false OutputIterators, and shall be convertible to Predicate's argument type. The input range shall not overlap with either of the output ranges.

- Effects: For each iterator i in [first,last), copies \*i to the output range beginning with out\_true if pred(\*i) is true, or to the output range beginning with out\_false otherwise.
- Returns: A pair p such that p.first is the end of the output range beginning at out\_true and p.second is the end of the output range beginning at out\_false.
- 15 Complexity: Exactly last first applications of pred.

- Requires: ForwardIterator's value type shall be convertible to Predicate's argument type. [first, last) shall be partitioned by pred, i.e. all elements that satisfy 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:  $\mathcal{O}(log(last first))$  applications of pred.

# 25.4 Sorting and related operations

[alg.sorting]

- All the operations in 25.4 have two versions: one that takes a function object of type Compare and one that uses an operator<.
- <sup>2</sup> 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

§ 25.4 859

```
— The induced relation is a strict total ordering. — end note
```

1

2

3

1

2

3

4

§ 25.4.1.3

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.
- 7 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an operator==, but an equivalence relation induced by the strict weak ordering. That is, two elements a and b are considered equivalent if and only if !(a < b) && !(b < a).

```
25.4.1
         Sorting
                                                                                             [alg.sort]
25.4.1.1
                                                                                                  [sort]
          sort
template < class Random Access Iterator >
  void sort(RandomAccessIterator first, RandomAccessIterator last);
template < class Random AccessIterator, class Compare >
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
     Effects: Sorts the elements in the range [first,last).
     Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The
     type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable
     (Table 22).
     Complexity: \mathcal{O}(N \log(N)) (where N == last - first) comparisons.
                                                                                           [stable.sort]
25.4.1.2 stable_sort
template<class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
template < class Random Access Iterator, class Compare >
  void stable sort(RandomAccessIterator first, RandomAccessIterator last,
                    Compare comp);
     Effects: Sorts the elements in the range [first,last).
     Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The
     type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable
     (Table 22).
     Complexity: It does at most N \log^2(N) (where N == last - first) comparisons; if enough extra
     memory is available, it is N \log(N).
     Remarks: Stable.
25.4.1.3 partial_sort
                                                                                          [partial.sort]
template < class Random AccessIterator >
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                     RandomAccessIterator last);
template < class Random AccessIterator, class Compare >
```

860

1

2

3

1

2

3

4

```
void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last,
                    Compare comp);
     Effects: Places the first middle - first sorted elements from the range [first,last) into the range
     [first,middle). The rest of the elements in the range [middle,last) are placed in an unspecified
     Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The
     type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable
     (Table 22).
     Complexity: It takes approximately (last - first) * log(middle - first) comparisons.
                                                                                 [partial.sort.copy]
25.4.1.4 partial_sort_copy
template < class InputIterator, class RandomAccessIterator>
 RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template < class InputIterator, class RandomAccessIterator,
         class Compare>
 RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
     Effects: Places the first min(last - first, result_last - result_first) sorted elements into the
     range [result_first,result_first + min(last - first, result_last - result_first)).
     Returns: The smaller of: result_last or result_first + (last - first).
     Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The
     type of *result_first shall satisfy the requirements of MoveConstructible (Table 20) and of Move-
     Assignable (Table 22).
     Complexity: Approximately (last - first) * log(min(last - first, result_last - result_-
     first)) comparisons.
                                                                                          [is.sorted]
25.4.1.5 is_sorted
template<class ForwardIterator>
 bool is_sorted(ForwardIterator first, ForwardIterator last);
     Returns: is_sorted_until(first, last) == last
template < class Forward Iterator, class Compare >
 bool is_sorted(ForwardIterator first, ForwardIterator last,
    Compare comp);
     Returns: is_sorted_until(first, last, comp) == last
template<class ForwardIterator>
 ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);
§ 25.4.1.5
                                                                                                 861
```

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 Random AccessIterator >

[alg.nth.element]

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 (17.6.3.2). The type of \*first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

3 Complexity: Linear on average.

## 25.4.3 Binary search

[alg.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

1

[lower.bound]

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.

3 Complexity: At most  $\log_2(last - first) + \mathcal{O}(1)$  comparisons.

§ 25.4.3.1 862

```
25.4.3.2 upper_bound
                                                                                       [upper.bound]
template < class ForwardIterator, class T>
 ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value);
template<class ForwardIterator, class T, class Compare>
 ForwardIterator
   upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
     Requires: The elements e of [first,last) shall be partitioned with respect to the expression! (value
     < e) or !comp(value, e).
     Returns: The furthermost iterator i in the range [first,last] such that for any iterator j in the
     range [first,i) the following corresponding conditions hold: !(value < *j) or comp(value, *j)
     == false.
     Complexity: At most \log_2(last - first) + \mathcal{O}(1) comparisons.
25.4.3.3 equal_range
                                                                                        [equal.range]
template < class ForwardIterator, class T>
 pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
                ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first,
                ForwardIterator last, const T& value,
                Compare comp);
     Requires: The elements e of [first,last) shall be partitioned with respect to the expressions e
     < value and !(value < e) or comp(e, value) and !comp(value, e). Also, for all elements e of
     [first, last), e < value shall imply !(value < e) or comp(e, value) shall imply !comp(value,
     e).
     Returns:
       make_pair(lower_bound(first, last, value),
                 upper_bound(first, last, value))
     or
       make_pair(lower_bound(first, last, value, comp),
                 upper_bound(first, last, value, comp))
     Complexity: At most 2 * \log_2(last - first) + \mathcal{O}(1) comparisons.
25.4.3.4 binary_search
                                                                                      [binary.search]
template<class ForwardIterator, class T>
 bool binary_search(ForwardIterator first, ForwardIterator last,
                     const T& value);
template < class Forward Iterator, class T, class Compare >
 bool binary_search(ForwardIterator first, ForwardIterator last,
                     const T& value, Compare comp);
§ 25.4.3.4
                                                                                                   863
```

1

2

```
Requires: The elements e of [first,last) are partitioned with respect to the expressions e < value and !(value < e) or comp(e, value) and !comp(value, e). Also, for all elements e of [first, last), e < value implies !(value < e) or comp(e, value) implies !comp(value, e).
```

- Returns: true if there is an iterator i in the range [first,last) that satisfies the corresponding conditions: !(\*i < value) && !(value < \*i) or comp(\*i, value) == false && comp(value, \*i) == false.
- 3 Complexity: At most log2(last first) +  $\mathcal{O}(1)$  comparisons.

1

2

3

4

5

```
25.4.4 Merge [alg.merge]
```

```
template < class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);
template < class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
   merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);
     Effects: Copies all the elements of the two ranges [first1,last1) and [first2,last2) into the range
     [result_result_last), where result_last is result + (last1 - first1) + (last2 - first2),
     such that the resulting range satisfies is_sorted(result, result_last) or is_sorted(result,
     result_last, comp), respectively.
     Requires: The ranges [first1,last1) and [first2,last2) shall be sorted with respect to operator<
     or comp. The resulting range shall not overlap with either of the original ranges.
     Returns: result + (last1 - first1) + (last2 - first2).
     Complexity: At most (last1 - first1) + (last2 - first2) - 1 comparisons.
     Remarks: Stable.
template < class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template < class BidirectionalIterator, class Compare >
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
```

- Effects: Merges two sorted consecutive ranges [first,middle) and [middle,last), putting the result of the merge into the range [first,last). The resulting range will be in non-decreasing order; that is, for every iterator i in [first,last) other than first, the condition \*i < \*(i 1) or, respectively, comp(\*i, \*(i 1)) will be false.
- Requires: The ranges [first,middle) and [middle,last) shall be sorted with respect to operator or comp. BidirectionalIterator shall satisfy the requirements of ValueSwappable (17.6.3.2). The

§ 25.4.4 864

type of \*first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

Complexity: When enough additional memory is available, (last - first) - 1 comparisons. If no additional memory is available, an algorithm with complexity  $N \log(N)$  (where N is equal to last - first) may be used.

9 Remarks: Stable.

1

1

#### 25.4.5 Set operations on sorted structures

[alg.set.operations]

This section defines all the basic set operations on sorted structures. They also work with multisets (23.4.7) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to multisets in a standard way by defining set\_union() to contain the maximum number of occurrences of every element, set intersection() to contain the minimum, and so on.

25.4.5.1 includes [includes]

25.4.5.2 set\_union [set.union]

Effects: Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

- 2 Requires: The resulting range shall not overlap with either of the original ranges.
- 3 Returns: The end of the constructed range.
- 4 Complexity: At most 2 \* ((last1 first1) + (last2 first2)) 1 comparisons.
- 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.

§ 25.4.5.2

```
25.4.5.3 set_intersection
                                                                                     [set.intersection]
template < class InputIterator1, class InputIterator2,
         class OutputIterator>
 OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
template < class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
     Effects: Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements
     that are present in both of the ranges.
     Requires: The resulting range shall not overlap with either of the original ranges.
     Returns: The end of the constructed range.
     Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 comparisons.
     Remarks: If [first1,last1) contains m elements that are equivalent to each other and [first2,
     last2) contains n elements that are equivalent to them, the first \min(m,n) elements shall be copied
     from the first range to the output range, in order.
25.4.5.4 set_difference
                                                                                       [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 [first1,last1) which are not present in the range [first2,
     last2) to the range beginning at result. The elements in the constructed range are sorted.
     Requires: The resulting range shall not overlap with either of the original ranges.
     Returns: The end of the constructed range.
     Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 comparisons.
     Remarks: If [first1,last1) contains m elements that are equivalent to each other and [first2,
     last2) contains n elements that are equivalent to them, the last \max(m-n,0) elements from [first1,
     last1) shall be copied to the output range.
```

1

3

4

5

2

3

4

5

25.4.5.5 set\_symmetric\_difference

§ 25.4.5.5

[set.symmetric.difference]

```
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);
        Effects: Copies the elements of the range [first1,last1) that are not present in the range [first2,
        last2), and the elements of the range [first2,last2) that are not present in the range [first1,
        last1) to the range beginning at result. The elements in the constructed range are sorted.
        Requires: The resulting range shall not overlap with either of the original ranges.
        Returns: The end of the constructed range.
        Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 comparisons.
        Remarks: If [first1,last1) contains m elements that are equivalent to each other and [first2,
        last2) contains n elements that are equivalent to them, then |m-n| of those elements shall be copied
        to the output range: the last m-n of these elements from [first1,last1) if m>n, and the last
        n-m of these elements from [first2,last2) if m < n.
                                                                                 [alg.heap.operations]
   25.4.6 Heap operations
A heap is a particular organization of elements in a range between two random access iterators [a,b). Its
   two key properties are:
    (1) There is no element greater than *a in the range and
    (2) *a may be removed by pop_heap(), or a new element added by push_heap(), in \mathcal{O}(\log(N)) time.
<sup>2</sup> These properties make heaps useful as priority queues.
  make_heap() converts a range into a heap and sort_heap() turns a heap into a sorted sequence.
                                                                                             [push.heap]
   25.4.6.1 push_heap
   template < class Random Access Iterator >
     void push_heap(RandomAccessIterator first, RandomAccessIterator last);
   template < class Random Access Iterator, class Compare >
     void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                    Compare comp);
        Effects: Places the value in the location last - 1 into the resulting heap [first,last).
        Requires: The range [first,last - 1) shall be a valid heap. The type of *first shall satisfy the
```

1

2

3

4

5

1

2

3

25.4.6.2 pop\_heap

§ 25.4.6.2 867

MoveConstructible requirements (Table 20) and the MoveAssignable requirements (Table 22).

[pop.heap]

Complexity: At most log(last - first) comparisons.

```
template<class RandomAccessIterator>
     void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
   template < class Random Access Iterator, class Compare >
     void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
1
        Requires: The range [first,last) shall be a valid non-empty heap. RandomAccessIterator shall sat-
        isfy the requirements of ValueSwappable (17.6.3.2). The type of *first shall satisfy the requirements
        of MoveConstructible (Table 20) and of MoveAssignable (Table 22).
        Effects: Swaps the value in the location first with the value in the location last - 1 and makes
        [first,last - 1) into a heap.
3
        Complexity: At most 2 * log(last - first) comparisons.
            make_heap
                                                                                            [make.heap]
   25.4.6.3
   template < class Random AccessIterator >
     void make_heap(RandomAccessIterator first, RandomAccessIterator last);
   template<class RandomAccessIterator, class Compare>
     void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                    Compare comp);
1
        Effects: Constructs a heap out of the range [first,last).
2
        Requires: The type of *first shall satisfy the MoveConstructible requirements (Table 20) and the
        MoveAssignable requirements (Table 22).
3
        Complexity: At most 3 * (last - first) comparisons.
                                                                                             [sort.heap]
   25.4.6.4
            sort_heap
   template < class Random AccessIterator >
     void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
   template < class Random Access Iterator, class Compare >
     void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                    Compare comp);
1
        Effects: Sorts elements in the heap [first,last).
        Requires: The range [first,last) shall be a valid heap. RandomAccessIterator shall satisfy the
        requirements of ValueSwappable (17.6.3.2). The type of *first shall satisfy the requirements of
        MoveConstructible (Table 20) and of MoveAssignable (Table 22).
3
        Complexity: At most N \log(N) comparisons (where N == last - first).
   25.4.6.5
             is_heap
                                                                                                [is.heap]
     template < class Random Access Iterator >
       bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
        Returns: is_heap_until(first, last) == last
     template<class RandomAccessIterator, class Compare>
       bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
2
        Returns: is_heap_until(first, last, comp) == last
```

§ 25.4.6.5

```
template<class RandomAccessIterator>
        RandomAccessIterator is heap until(RandomAccessIterator first, RandomAccessIterator last);
      template < class Random Access Iterator, class Compare >
        RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator 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 a heap.
         Complexity: Linear.
   25.4.7 Minimum and maximum
                                                                                          [alg.min.max]
   template<class T> const T& min(const T& a, const T& b);
   template < class T, class Compare >
      const T& min(const T& a, const T& b, Compare comp);
1
         Requires: Type T is LessThanComparable (Table 18).
2
         Returns: The smaller value.
3
         Remarks: Returns the first argument when the arguments are equivalent.
   template<class T>
     T min(initializer_list<T> t);
   template < class T, class Compare >
      T min(initializer_list<T> t, Compare comp);
4
         Requires: T is LessThanComparable and CopyConstructible and t.size() > 0.
5
         Returns: The smallest value in the initializer list.
6
         Remarks: Returns a copy of the leftmost argument when several arguments are equivalent to the
         smallest.
   template<class T> const T& max(const T& a, const T& b);
   template < class T, class Compare >
      const T& max(const T& a, const T& b, Compare comp);
7
         Requires: Type T is LessThanComparable (Table 18).
8
         Returns: The larger value.
9
         Remarks: Returns the first argument when the arguments are equivalent.
   template<class T>
     T max(initializer_list<T> t);
   template < class T, class Compare >
      T max(initializer_list<T> t, Compare comp);
10
         Requires: T is LessThanComparable and CopyConstructible and t.size() > 0.
11
         Returns: The largest value in the initializer_list.
12
         Remarks: Returns a copy of the leftmost argument when several arguments are equivalent to the
         largest.
```

§ 25.4.7

```
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);
13
         Requires: Type T shall be LessThanComparable (Table 18).
14
         Returns: pair<const T&, const T&>(b, a) if b is smaller than a, and pair<const T&, const
         T&>(a, b) otherwise.
15
         Remarks: Returns pair<const T&, const T&>(a, b) when the arguments are equivalent.
16
         Complexity: Exactly one comparison.
   template<class T>
     pair<T, T> minmax(initializer_list<T> t);
   template < class T, class Compare >
      pair<T, T> minmax(initializer_list<T> t, Compare comp);
17
         Requires: T is LessThanComparable and CopyConstructible and t.size() > 0.
18
         Returns: pair<T, T>(x, y), where x has the smallest and y has the largest value in the initializer
         list.
19
         Remarks: x is a copy of the leftmost argument when several arguments are equivalent to the smallest.
         y is a copy of the rightmost argument when several arguments are equivalent to the largest.
20
         Complexity: At most (3/2) * t.size() applications of the corresponding predicate.
   template < class ForwardIterator>
     ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
   template < class Forward Iterator, class Compare >
     ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                                Compare comp);
21
         Returns: The first iterator i in the range [first,last) such that for any iterator j in the range
         [first,last) the following corresponding conditions hold: !(*j < *i) or comp(*j, *i) == false.
         Returns last if first == last.
22
         Complexity: Exactly max((last - first) - 1, 0) applications of the corresponding comparisons.
   template<class ForwardIterator>
     ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
   template < class Forward Iterator, class Compare >
     ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                Compare comp);
23
         Returns: The first iterator i in the range [first,last) such that for any iterator j in the range
         [first,last) the following corresponding conditions hold: !(*i < *j) or comp(*i, *j) == false.
         Returns last if first == last.
24
         Complexity: Exactly max((last - first) - 1, 0) applications of the corresponding comparisons.
```

§ 25.4.7

```
template<class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
  minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
```

Returns: make\_pair(first, first) if [first,last) is empty, otherwise 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.

Complexity: At most  $max(\lfloor \frac{3}{2}(N-1)\rfloor, 0)$  applications of the corresponding predicate, where N is distance(first, last).

# 25.4.8 Lexicographical comparison

25

26

1

[alg.lex.comparison]

- Returns: true if the sequence of elements defined by the range [first1,last1) is lexicographically less than the sequence of elements defined by the range [first2,last2) and false otherwise.
- 2 Complexity: At most 2\*min((last1 first1), (last2 first2)) applications of the corresponding comparison.
- 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;</pre>
```

4 Remarks: An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

## 25.4.9 Permutation generators

[alg.permutation.generators]

§ 25.4.9

Effects: Takes a sequence defined by the range [first,last) and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to operator< or comp. If such a permutation exists, it returns true. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns false.

- 2 Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (17.6.3.2).
- 3 Complexity: At most (last first)/2 swaps.

- 4 Effects: Takes a sequence defined by the range [first,last) and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to operator< or comp.
- Returns: true if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns false.
- 6 Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (17.6.3.2).
- 7 Complexity: At most (last first)/2 swaps.

# 25.5 C library algorithms

[alg.c.library]

Table 113 describes some of the contents of the header <cstdlib>.

Table 113 — Header <cstdlib> synopsis

Type	Name(s)	
Type:	size_t	
Functions:	bsearch	qsort

- The contents are the same as the Standard C library header <stdlib.h> with the following exceptions:
- <sup>3</sup> The function signature:

both of which have the same behavior as the original declaration.

<sup>4</sup> The function signature:

§ 25.5 872

both of which have the same behavior as the original declaration. The behavior is undefined unless the objects in the array pointed to by base are of trivial type.

[Note: Because the function argument compar() may throw an exception, bsearch() and qsort() are allowed to propagate the exception (17.6.5.12). — end note]

SEE ALSO: ISO C 7.10.5.

§ 25.5

# 26 Numerics library

# [numerics]

26.1 General [numerics.general]

This Clause describes components that C++ programs may use to perform seminumerical operations.

<sup>2</sup> 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 114.

	Subclause	Header(s)
26.2	Requirements	
26.3	Floating-Point Environment	<cfenv></cfenv>
26.4	Complex Numbers	<complex></complex>
26.5	Random number generation	<random></random>
26.6	Numeric arrays	<valarray></valarray>
26.7	Generalized numeric operations	<numeric></numeric>
26.8	C library	<cmath></cmath>
		<ctgmath></ctgmath>
		<tgmath.h></tgmath.h>
		<cstdlib></cstdlib>

Table 114 — Numerics library summary

# 26.2 Numeric type requirements

## [numeric.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: 272
  - 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.

§ 26.2 874

<sup>272)</sup> 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&.
- If any operation on T throws an exception the effects are undefined.
- In addition, many member and related functions of valarray<T> can be successfully instantiated and will exhibit well-defined behavior if and only if T satisfies additional requirements specified for each such member or related function.
- [Example: It is valid to instantiate valarray<complex>, but operator>() will not be successfully instantiated for valarray<complex> operands, since complex does not have any ordering operators. — end example

#### The floating-point environment 26.3

[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);
The header also defines the macros:
```

FE\_ALL\_EXCEPT FE\_DIVBYZERO FE\_INEXACT FE\_INVALID FE\_OVERFLOW

§ 26.3.1 875

```
FE_UNDERFLOW
FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD
FE_DFL_ENV
```

- <sup>2</sup> The header defines all functions, types, and macros the same as Clause 7.6 of the C standard.
- 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]
- <sup>4</sup> 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]

- <sup>1</sup> The header <complex> defines a class template, and numerous functions for representing and manipulating complex numbers.
- The effect of instantiating the template complex for any type other than float, double, or long double is unspecified. The specializations complex<float>, complex<double>, and complex<long double> are literal types (3.9).
- <sup>3</sup> 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.
- <sup>4</sup> 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>
    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 T&, const complex<T>&);
  template<class T> complex<T> operator-(
    const complex<T>&, const complex<T>&);
```

§ 26.4.1

```
template<class T> complex<T> operator-(const complex<T>&, const T&);
template<class T> complex<T> operator-(const T&, const complex<T>&);
template<class T> complex<T> operator*(
  const complex<T>&, const complex<T>&);
template<class T> complex<T> operator*(const complex<T>&, const T&);
template<class T> complex<T> operator*(const T&, const complex<T>&);
template<class T> complex<T> operator/(
  const complex<T>&, const complex<T>&);
template<class T> complex<T> operator/(const complex<T>&, const T&);
template<class T> complex<T> operator/(const T&, const complex<T>&);
template<class T> complex<T> operator+(const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&);
template<class T> bool operator==(
  const complex<T>&, const complex<T>&);
template<class T> bool operator==(const complex<T>&, const T&);
template<class T> bool operator==(const T&, const complex<T>&);
template<class T> bool operator!=(const complex<T>&, const complex<T>&);
template<class T> bool operator!=(const complex<T>&, const T&);
template<class T> bool operator!=(const T&, const complex<T>&);
template < class T, class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>&, complex<T>&);
template < class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>&, const complex<T>&);
// 26.4.7, values:
template<class T> T real(const complex<T>&);
template<class T> T imag(const complex<T>&);
template<class T> T abs(const complex<T>&);
template<class T> T arg(const complex<T>&);
template<class T> T norm(const complex<T>&);
template<class T> complex<T> conj(const complex<T>&);
template <class T> complex<T> proj(const complex<T>&);
template<class T> complex<T> polar(const T&, const T& = 0);
// 26.4.8, transcendentals:
template<class T> complex<T> acos(const complex<T>&);
template<class T> complex<T> asin(const complex<T>&);
template<class T> complex<T> atan(const complex<T>&);
template<class T> complex<T> acosh(const complex<T>&);
template<class T> complex<T> asinh(const complex<T>&);
template<class T> complex<T> atanh(const complex<T>&);
template<class T> complex<T> cos (const complex<T>&);
```

§ 26.4.1

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
                                                                                         [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>&);
    };
  }
The class complex describes an object that can store the Cartesian components, real() and imag(), of a
complex number.
26.4.3 complex specializations
                                                                                 [complex.special]
  namespace std {
    template<> class complex<float> {
    public:
      typedef float value_type;
§ 26.4.3
                                                                                                 878
```

```
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> {
 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 complex<long double> {
 typedef long double value_type;
```

§ 26.4.3

```
constexpr complex(long double re = 0.0L, long double im = 0.0L);
      constexpr complex(const complex<float>&);
      constexpr complex(const complex<double>&);
      constexpr long double real();
      void real(long double);
      constexpr 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
                                                                              [complex.members]
template<class T> complex(const T% re = T(), const T% im = T());
     Effects: Constructs an object of class complex.
     Postcondition: real() == re && imag() == 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.member.ops]
complex<T>& operator+=(const T& rhs);
     Effects: Adds the scalar value rhs to the real part of the complex value *this and stores the result in
     the real part of *this, leaving the imaginary part unchanged.
     Returns: *this.
§ 26.4.5
                                                                                                 880
```

1

```
complex<T>& operator-=(const T& rhs);
3
         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.
4
         Returns: *this.
   complex<T>& operator*=(const T& rhs);
5
         Effects: Multiplies the scalar value rhs by the complex value *this and stores the result in *this.
6
         Returns: *this.
   complex<T>& operator/=(const T& rhs);
7
         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);
9
         Effects: Adds the complex value rhs to the complex value *this and stores the sum in *this.
10
         Returns: *this.
   complex<T>& operator-=(const complex<T>& rhs);
11
         Effects: Subtracts the complex value rhs from the complex value *this and stores the difference in
         *this.
12
         Returns: *this.
   complex<T>& operator*=(const complex<T>& rhs);
13
         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);
14
         Effects: Divides the complex value rhs into the complex value *this and stores the quotient in *this.
15
         Returns: *this.
   26.4.6 complex non-member operations
                                                                                           [complex.ops]
   template<class T> complex<T> operator+(const complex<T>& lhs);
1
         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);
   § 26.4.6
                                                                                                        881
```

```
3
         Returns: complex<T>(lhs) += rhs.
   template<class T> complex<T> operator-(const complex<T>& lhs);
4
         Remarks: unary operator.
5
         Returns: complex<T>(-lhs.real(),-lhs.imag()).
   template<class T>
     complex<T> operator-(const complex<T>& lhs, const complex<T>& rhs);
   template<class T> complex<T> operator-(const complex<T>& lhs, const T& rhs);
   template<class T> complex<T> operator-(const T& lhs, const complex<T>& rhs);
6
         Returns: complex<T>(lhs) -= rhs.
   template<class T>
     \label{lem:complexT} $$ \operatorname{complexT>\& hs, const complexT>\& rhs);} $$
   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);
7
         Returns: complex<T>(lhs) *= rhs.
   template<class T>
     complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);
   template<class T> complex<T> operator/(const complex<T>& lhs, const T& rhs);
   template<class T> complex<T> operator/(const T& lhs, const complex<T>& rhs);
         Returns: complex<T>(lhs) /= rhs.
   template<class T>
     bool operator==(const complex<T>& lhs, const complex<T>& rhs);
   template<class T> bool operator==(const complex<T>& lhs, const T& rhs);
   template<class T> bool operator==(const T& lhs, const complex<T>& rhs);
9
         Returns: lhs.real() == rhs.real() && lhs.imag() == rhs.imag().
10
         Remarks: The imaginary part is assumed to be T(), or 0.0, for the T arguments.
   template<class T>
     bool operator!=(const complex<T>& lhs, const complex<T>& rhs);
   template<class T> bool operator!=(const complex<T>& lhs, const T& rhs);
   template<class T> bool operator!=(const T& lhs, const complex<T>& rhs);
11
         Returns: rhs.real() != lhs.real() || rhs.imag() != lhs.imag().
   template < class T, class charT, class traits>
   basic_istream<charT, traits>&
   operator>>(basic_istream<charT, traits>& is, complex<T>& x);
```

§ 26.4.6

```
Effects: Extracts a complex number x of the form: u, (u), or (u,v), where u is the real part and v is the imaginary part (27.7.2.2).
```

13 Requires: The input values shall be convertible to T.

If bad input is encountered, calls is.setstate(ios\_base::failbit) (which may throw ios::failure (27.5.5.4)).

- 14 Returns: is.
- Remarks: This extraction is performed as a series of simpler extractions. Therefore, the skipping of whitespace is specified to be the same for each of the simpler extractions.

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& o, const complex<T>& x);
```

16 Effects: inserts the complex number x onto the stream o as if it were implemented as follows:

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>& o, const complex<T>& x) {
    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();
}</pre>
```

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::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

[complex.value.ops]

```
template<class T> T real(const complex<T>& x);

Returns: x.real().

template<class T> T imag(const complex<T>& x);

Returns: x.imag().

template<class T> T abs(const complex<T>& x);

Returns: The magnitude of x.

template<class T> T arg(const complex<T>& x);

Returns: The phase angle of x, or atan2(imag(x), real(x)).

template<class T> T norm(const complex<T>& x);
```

§ 26.4.7

```
5
         Returns: The squared magnitude of x.
   template<class T> complex<T> conj(const complex<T>& x);
6
         Returns: The complex conjugate of x.
   template<class T> complex<T> proj(const complex<T>& x);
7
         Returns: The projection of x onto the Riemann sphere.
8
         Remarks: Behaves the same as the C function cproj, defined in 7.3.9.4.
   template<class T> complex<T> polar(const T& rho, const T& theta = 0);
9
         Returns: The complex value corresponding to a complex number whose magnitude is rho and whose
         phase angle is theta.
                                                                            [complex.transcendentals]
   26.4.8
             complex transcendentals
   template<class T> complex<T> acos(const complex<T>& x);
1
         Returns: The complex arc cosine of x.
2
         Remarks: Behaves the same as C function cacos, defined in 7.3.5.1.
   template<class T> complex<T> asin(const complex<T>& x);
3
         Returns: The complex arc sine of x.
         Remarks: Behaves the same as C function casin, defined in 7.3.5.2.
   template<class T> complex<T> atan(const complex<T>& x);
5
         Returns: The complex arc tangent of x.
6
         Remarks: Behaves the same as C function catan, defined in 7.3.5.3.
   template<class T> complex<T> acosh(const complex<T>& x);
7
         Returns: The complex arc hyperbolic cosine of x.
8
         Remarks: Behaves the same as C function cacosh, defined in 7.3.6.1.
   template<class T> complex<T> asinh(const complex<T>& x);
9
         Returns: The complex arc hyperbolic sine of x.
10
         Remarks: Behaves the same as C function casinh, defined in 7.3.6.2.
   template<class T> complex<T> atanh(const complex<T>& x);
11
         Returns: The complex arc hyperbolic tangent of x.
12
         Remarks: Behaves the same as C function catanh, defined in 7.3.6.3.
   § 26.4.8
                                                                                                       884
```

```
template<class T> complex<T> cos(const complex<T>& x);
13
         Returns: The complex cosine of x.
    template<class T> complex<T> cosh(const complex<T>& x);
14
         Returns: The complex hyperbolic cosine of x.
    template<class T> complex<T> exp(const complex<T>& x);
15
         Returns: The complex base e exponential of x.
    template<class T> complex<T> log(const complex<T>& x);
16
         Remarks: the branch cuts are along the negative real axis.
17
         Returns: The complex natural (base e) logarithm of x, in the range of a strip mathematically un-
         bounded 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);
18
         Remarks: the branch cuts are along the negative real axis.
19
         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);
20
         Remarks: the branch cuts are along the negative real axis.
21
         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);
22
         Returns: The complex sine of x.
    template<class T> complex<T> sinh (const complex<T>& x);
23
         Returns: The complex hyperbolic sine of x.
    template<class T> complex<T> sqrt (const complex<T>& x);
24
         Remarks: the branch cuts are along the negative real axis.
25
         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.
```

§ 26.4.8

```
template<class T> complex<T> tan (const complex<T>& x);
```

Returns: The complex tangent of x.

26

27

template<class T> complex<T> tanh (const complex<T>& x);

Returns: The complex hyperbolic tangent of x.

#### 26.4.9 Additional overloads

[cmplx.over]

The following function templates shall have additional overloads:

arg norm conj proj imag real

- <sup>2</sup> 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>.
  - Otherwise, if the argument has type double or an integer type, then it is effectively cast to complex< double>.
  - 3. Otherwise, if the argument has type float, then it is effectively cast to complex<float>.
- Function template pow shall have additional overloads sufficient to ensure, for a call with at least one argument of type complex<T>:
  - 1. If either argument has type complex<long double> or type long double, then both arguments are effectively cast to complex<long double>.
  - 2. Otherwise, if either argument has type complex<double>, double, or an integer type, then both arguments are effectively cast to complex<double>.
  - 3. Otherwise, if either argument has type complex<float> or float, then both arguments are effectively cast to complex<float>.

### 26.4.10 Header <ccomplex>

[ccmplx]

The header behaves as if it simply includes the header <complex>.

#### 26.5 Random number generation

[rand]

- This subclause defines a facility for generating (pseudo-)random numbers.
- In addition to a few utilities, four categories of entities are described: uniform random number generators, random number engines, random number engine adaptors, and random number distributions. These categorizations are applicable to types that satisfy the corresponding requirements, to objects instantiated from such types, and to templates producing such types when instantiated. [Note: These entities are specified in such a way as to permit the binding of any uniform random number generator object e as the argument to any random number distribution object d, thus producing a zero-argument function object such as given by bind(d,e). end note]
- Each of the entities specified via this subclause has an associated arithmetic type (3.9.1) identified as result\_type. With T as the result\_type thus associated with such an entity, that entity is characterized:
  - a) as boolean or equivalently as boolean-valued, if T is bool;
  - b) otherwise as integral or equivalently as integer-valued, if numeric\_limits<T>::is\_integer is true;

§ 26.5

c) otherwise as *floating* or equivalently as *real-valued*.

If integer-valued, an entity may optionally be further characterized as *signed* or *unsigned*, according to numeric limits<T>::is signed.

- 4 Unless otherwise specified, all descriptions of calculations in this subclause use mathematical real numbers.
- Throughout this subclause, the operators bitand, bitor, and xor denote the respective conventional bitwise operations. Further:
  - a) the operator rshift denotes a bitwise right shift with zero-valued bits appearing in the high bits of the result, and
  - b) the operator  $\mathsf{lshift}_w$  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 Sseq is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of seed sequence (26.5.1.2).
  - b) that has a template type parameter named URNG is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of uniform random number generator (26.5.1.3).
  - c) that has a template type parameter named Engine is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of random number engine (26.5.1.4).
  - d) that has a template type parameter named RealType is undefined unless the corresponding template argument is cv-unqualified and is one of float, double, or long double.
  - e) that has a template type parameter named IntType is undefined unless the corresponding template argument is cv-unqualified and is one of short, int, long, long long, unsigned short, unsigned int, unsigned long, or unsigned long long.
  - f) that has a template type parameter named UIntType is undefined unless the corresponding template argument is cv-unqualified and is one of unsigned short, unsigned int, unsigned long, or unsigned long long.
- Throughout this subclause 26.5, phrases of the form "x is an iterator of a specific kind" shall be interpreted as equivalent to the more formal requirement that "x is a value of a type satisfying the requirements of the specified iterator type."
- 3 Throughout this subclause 26.5, any constructor that can be called with a single argument and that satisfies a requirement specified in this subclause shall be declared explicit.

#### 26.5.1.2 Seed sequence requirements

[rand.req.seedseq]

- A seed sequence is an object that consumes a sequence of integer-valued data and produces a requested number of unsigned integer values i,  $0 \le i < 2^{32}$ , 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]
- <sup>2</sup> A class S satisfies the requirements of a seed sequence if the expressions shown in Table 115 are valid and have the indicated semantics, and if S also satisfies all other requirements of this section 26.5.1.2. In that Table and throughout this section:
  - a) T is the type named by S's associated result type;
  - b) q is a value of S and r is a possibly const value of S;

§ 26.5.1.2

- c) ib and ie are input iterators with an unsigned integer value\_type of at least 32 bits;
- d) rb and re are mutable random access iterators with an unsigned integer value\_type of at least 32 bits:
- e) ob is an output iterator; and
- f) il is a value of initializer\_list<T>.

Table 115 — Seed sequence requirements

Expression	Return type	${ m Pre/post\text{-}condition}$	Complexity
S::result_type	T	T is an unsigned integer	compile-time
		type $(3.9.1)$ of at least 32 bits.	
S()		Creates a seed sequence with	constant
		the same initial state as all	
		other default-constructed seed	
		sequences of type S.	
S(ib,ie)		Creates a seed sequence having	$\mathscr{O}(\mathtt{ie}-\mathtt{ib})$
		internal state that depends on	
		some or all of the bits of the	
		supplied sequence [ib, ie).	
S(il)		Same as S(il.begin(),	same as
		il.end()).	S(il.begin(),
			il.end())
q.generate(rb,re)	void	Does nothing if rb == re.	$\mathscr{O}(\mathtt{re}-\mathtt{rb})$
		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.	
r.size()	size_t	The number of 32-bit units that	constant
		would be copied by a call to	
		r.param.	
r.param(ob)	void	Copies to the given destination	$\mathscr{O}(\mathtt{r.size()})$
		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.	

# ${\bf 26.5.1.3} \quad {\bf Uniform\ random\ number\ generator\ requirements}$

[rand.req.urng]

§ 26.5.1.3

A uniform random number generator g of type G is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. [Note: The degree to which g's results approximate the ideal is often determined statistically. —end note]

<sup>2</sup> A class **G** satisfies the requirements of a *uniform random number generator* if the expressions shown in Table 116 are valid and have the indicated semantics, and if **G** also satisfies all other requirements of this section 26.5.1.3. In that Table and throughout this section:

- a) T is the type named by G's associated result\_type, and
- b) g is a value of G.

Table 116 — Uniform random number generator requirements

Expression	Return type	${ m Pre/post\text{-}condition}$	Complexity
G::result_type	T	T is an unsigned integer	compile-time
		type $(3.9.1)$ .	
g()	T	Returns a value in the closed	amortized
		interval [G::min(), G::max()].	constant
G::min()	Т	Denotes the least value	compile-time
		potentially returned by	
		operator().	
G::max()	Т	Denotes the greatest value	compile-time
		potentially returned by	
		operator().	

The following relation shall hold: G::min() < G::max().

#### 26.5.1.4 Random number engine requirements

[rand.req.eng]

- 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  $i \ge 0$ . 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>>.
- <sup>3</sup> 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.
- <sup>4</sup> A class E that satisfies the requirements of a uniform random number generator (26.5.1.3) also satisfies the requirements of a *random number engine* if the expressions shown in Table 117 are valid and have the indicated semantics, and if E also satisfies all other requirements of this section 26.5.1.4. In that Table and throughout this section:
  - a) T is the type named by E's associated 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 Ivalue 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.

§ 26.5.1.4 889

Table 117 — Random number engine requirements

Expression	Return type	Pre/post-condition	Complexity
E()		Creates an engine with the same initial state as all other default-constructed engines of type E.	$\mathscr{O}(\text{size of state})$
E(x)		Creates an engine that compares equal to x.	$\mathcal{O}(\text{size of state})$
E(s)		Creates an engine with initial state determined by s.	$\mathscr{O}(\text{size of state})$
E(q) <sup>273</sup>		Creates an engine with an initial state that depends on a sequence produced by one call to q.generate.	same as complexity of q.generate called on a sequence whose length is size of state
e.seed()	void	post: e == E().	same as E()
e.seed(s)	void	post: e == E(s).	same as E(s)
e.seed(q)	void	post: $e == E(q)$ .	same as E(q)
e()	Т	Advances e's state $e_i$ to $e_{i+1}$ = $TA(e_i)$ and returns $GA(e_i)$ .	per Table 116
e.discard(z) <sup>274</sup>	void	Advances e's state $e_i$ to $e_{i+z}$ by any means equivalent to z consecutive calls $e()$ .	no worse than the complexity of z consecutive calls e()
x == y	bool	This operator is an equivalence relation. With $S_x$ and $S_y$ as the infinite sequences of values that would be generated by repeated future calls to $x()$ and $y()$ , respectively, returns true if $S_x = S_y$ ; else returns false.	$\mathscr{O}(\mathrm{size}\ \mathrm{of}\ \mathrm{state})$
x != y	bool	!(x == y).	$\mathcal{O}(\text{size of state})$
os << x	reference to the type of os	With os. fmtflags set to iosbase::dec ios_base::left and the fill character set to the space character, writes to os the textual representation of x's current state. In the output, adjacent numbers are separated by one or more space characters.  post: The os. fmtflags and fill character are unchanged.	$\mathscr{O}(\mathrm{size}\ \mathrm{of}\ \mathrm{state})$

<sup>273)</sup> This constructor (as well as the subsequent corresponding seed() function) may be particularly useful to applications requiring a large number of independent random sequences.

§ 26.5.1.4 890

<sup>274)</sup> 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  ${\bf z}$  consecutive calls  ${\bf e}()$ .

Expression	Return type	Pre/post-condition	Complexity
is >> v	reference to the type of is	With is.fmtflags set to ios_base::dec, sets v's state as determined by reading its textual representation from is. If bad input is encountered, ensures that v's state is unchanged by the operation and calls is.setstate(ios::failbit) (which may throw ios::failure [27.5.5.4]). If a textual representation written via os << x was subsequently read via is >> v, then x == v provided that there have been no intervening invocations of x or of v. pre: is provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of is, and whose type's template specialization arguments charT and traits were respectively the same as those of is. post: The is.fmtflags are unchanged.	$\mathscr{O}( ext{size of state})$

E shall meet the requirements of CopyConstructible (Table 21) and CopyAssignable (Table 23) 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

[rand.req.adapt]

- A random number engine adaptor (commonly shortened to adaptor) a of type A is a random number engine that takes values produced by some other random number engine, and applies an algorithm to those values in order to deliver a sequence of values with different randomness properties. An engine b of type B adapted in this way is termed a base engine in this context. The expression a.base() shall be valid and shall return a const reference to a's base engine.
- The requirements of a random number engine type shall be interpreted as follows with respect to a random number engine adaptor type.

A::A();

3 Effects: The base engine is initialized as if by its default constructor.

bool operator==(const A& a1, const A& a2);

4 Returns: true if al's base engine is equal to al's base engine. Otherwise returns false.

A::A(result\_type s);

§ 26.5.1.5

- <sup>10</sup> 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

[rand.req.dist]

- 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 118 are valid and have the indicated semantics, and if D and its associated types also satisfy all other requirements of this section 26.5.1.6. In that Table and throughout this section,
  - a) T is the type named by D's associated result\_type;
  - b) P is the type named by D's associated param\_type;
  - c) d is a value of D, and x and y are (possibly const) values of D;
  - d) glb and lub are values of T respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by d's operator(), as determined by the current values of d's parameters;

§ 26.5.1.6 892

- e) p is a (possibly const) value of P;
- f) g, g1, and g2 are lvalues of a type satisfying the requirements of a uniform random number generator [26.5.1.3];
- g) os is an lvalue of the type of some class template specialization basic\_ostream<charT, traits>; and
- h) is is an lvalue of the type of some class template specialization basic\_istream<charT, traits>; where charT and traits are constrained according to Clauses 21 and 27.

Table 118 — Random number distribution requirements

Expression	Return type	Pre/post-condition	Complexity
D::result_type	Т	T is an arithmetic type $(3.9.1)$ .	compile-time
D::param_type	P		compile-time
D()		Creates a distribution whose behavior is indistinguishable from that of any other newly default-constructed distribution of type D.	constant
D(p)		Creates a distribution whose behavior is indistinguishable from that of a distribution newly constructed directly from the values used to construct p.	same as p's construction
d.reset()	void	Subsequent uses of d do not depend on values produced by any engine prior to invoking reset.	constant
x.param()	Р	Returns a value p such that D(p).param() == p.	no worse than the complexity of D(p)
d.param(p)	void	post: d.param() == p.	no worse than the complexity of D(p)
d(g)	T	With $p = d.param()$ , the sequence of numbers returned by successive invocations with the same object g is randomly distributed according to the associated $p(z   \{p\})$ or $P(z_i   \{p\})$ function.	amortized constant number of invocations of g
d(g,p)	T	The sequence of numbers returned by successive invocations with the same objects $\mathbf{g}$ and $\mathbf{p}$ is randomly distributed according to the associated $p(z \mid \{\mathbf{p}\})$ or $P(z_i \mid \{\mathbf{p}\})$ function.	amortized constant number of invocations of g
x.min()	T	Returns glb.	constant
x.max()	T	Returns lub.	constant

§ 26.5.1.6

Expression	Return type	Pre/post-condition	Complexity
х == у	bool	This operator is an equivalence	constant
		relation. Returns true if	
		<pre>x.param() == y.param() and</pre>	
		$S_1 = S_2$ , where $S_1$ and $S_2$ are	
		the infinite sequences of values	
		that would be generated,	
		respectively, by repeated future	
		calls to $x(g1)$ and $y(g2)$	
		whenever $g1 == g2$ . Otherwise	
		returns false.	
x != y	bool	!(x == y).	same as x ==
			у.
os << x	reference to the type of	Writes to os a textual	
	os	representation for the	
		parameters and the additional	
		internal data of x.	
		post: The os. fmtflags and fill	
		character are unchanged.	
is >> d	reference to the type of	Restores from is the	
	is	parameters and additional	
		internal data of the lvalue d. If	
		bad input is encountered,	
		ensures that d is unchanged by	
		the operation and calls	
		is.setstate(ios::failbit)	
		(which may throw	
		ios::failure [27.5.5.4]).	
		pre: is provides a textual	
		representation that was	
		previously written using an os	
		whose imbued locale and whose	
		type's template specialization	
		arguments charT and traits	
		were the same as those of is.	
		post: The is. fmtflags are	
		unchanged.	

- <sup>4</sup> D shall satisfy the requirements of CopyConstructible (Table 21) and CopyAssignable (Table 23) types.
- The sequence of numbers produced by repeated invocations of d(g) shall be independent of any invocation of os << d or of any const member function of D between any of the invocations d(g).
- <sup>6</sup> If a textual representation is written using os << x and that representation is restored into the same or a different object y of the same type using is >> y, repeated invocations of y(g) shall produce the same sequence of numbers as would repeated invocations of x(g).
- 7 It is unspecified whether D::param\_type is declared as a (nested) class or via a typedef. In this subclause 26.5, declarations of D::param\_type are in the form of typedefs for convenience of exposition only.
- 8 P shall satisfy the requirements of CopyConstructible (Table 21), CopyAssignable (Table 23), and EqualityComparable (Table 17) types.
- <sup>9</sup> For each of the constructors of D taking arguments corresponding to parameters of the distribution, P shall have a corresponding constructor subject to the same requirements and taking arguments identical

§ 26.5.1.6 894

in number, type, and default values. Moreover, for each of the member functions of D that return values corresponding to parameters of the distribution, P shall have a corresponding member function with the identical name, type, and semantics.

10 P shall have a declaration of the form

```
typedef D distribution_type;
```

## 26.5.2 Header <random> synopsis

[rand.synopsis]

```
#include <initializer_list>
namespace std {
 // 26.5.3.1, class template linear_congruential_engine
 template < class UIntType, UIntType a, UIntType c, UIntType m>
   class linear_congruential_engine;
 // 26.5.3.2, class template mersenne_twister_engine
 template < class UIntType, size_t w, size_t n, size_t m, size_t r,
          UIntType a, size_t u, UintType d, size_t s,
          UIntType b, size_t t,
          UIntType c, size_t l, UintType f>
   class mersenne_twister_engine;
 // 26.5.3.3, class template subtract_with_carry_engine
 template<class UIntType, size_t w, size_t s, size_t r>
   class subtract_with_carry_engine;
 // 26.5.4.2, class template discard_block_engine
 template<class Engine, size_t p, size_t r>
   class discard_block_engine;
 // 26.5.4.3, class template independent_bits_engine
 template<class Engine, size_t w, class UIntType>
   class independent_bits_engine;
 // 26.5.4.4, class template shuffle_order_engine
 template < class Engine, size_t k>
   class shuffle_order_engine;
 // 26.5.5, engines and engine adaptors with predefined parameters
 typedef see below minstd_rand0;
 typedef see below minstd_rand;
 typedef see below mt19937;
 typedef see below mt19937_64;
 typedef see below ranlux24_base;
 typedef see below ranlux48_base;
 typedef see below ranlux24;
 typedef see below ranlux48;
 typedef see below knuth_b;
 typedef see below default_random_engine;
 // 26.5.6, class random_device
 class random_device;
```

§ 26.5.2 895

```
// 26.5.7.1, class seed_seq
class seed_seq;
// 26.5.7.2, function template generate_canonical
template < class RealType, size_t bits, class URNG>
  RealType generate_canonical(URNG& g);
// 26.5.8.2.1, class template uniform_int_distribution
template<class IntType = int>
  class uniform_int_distribution;
// 26.5.8.2.2, class template uniform_real_distribution
template<class RealType = double>
  class uniform_real_distribution;
// 26.5.8.3.1, class bernoulli_distribution
class bernoulli_distribution;
// 26.5.8.3.2, class template binomial_distribution
template < class IntType = int>
  class binomial_distribution;
// 26.5.8.3.3, class template geometric_distribution
template < class IntType = int>
  class geometric_distribution;
// 26.5.8.3.4, class template negative_binomial_distribution
template<class IntType = int>
  class negative_binomial_distribution;
// 26.5.8.4.1, class template poisson_distribution
template < class IntType = int>
  class poisson_distribution;
// 26.5.8.4.2, class template exponential_distribution
template < class RealType = double >
  class exponential_distribution;
// 26.5.8.4.3, class template gamma_distribution
template < class RealType = double >
  class gamma_distribution;
// 26.5.8.4.4, class template weibull_distribution
template < class RealType = double >
  class weibull_distribution;
// 26.5.8.4.5, class template extreme_value_distribution
template<class RealType = double>
  class extreme_value_distribution;
// 26.5.8.5.1, class template normal_distribution
template < class RealType = double >
  class normal_distribution;
// 26.5.8.5.2, class template lognormal_distribution
```

§ 26.5.2

```
template < class RealType = double >
   class lognormal_distribution;
 // 26.5.8.5.3, class template chi_squared_distribution
 template < class RealType = double >
   class chi_squared_distribution;
 // 26.5.8.5.4, class template cauchy distribution
 template < class RealType = double >
   class cauchy_distribution;
 // 26.5.8.5.5, class template fisher_f_distribution
 template<class RealType = double>
   class fisher_f_distribution;
 // 26.5.8.5.6, class template student_t_distribution
 template < class RealType = double >
   class student_t_distribution;
 // 26.5.8.6.1, class template discrete distribution
 template<class IntType = int>
   class discrete_distribution;
 // 26.5.8.6.2, class template piecewise_constant_distribution
 template < class RealType = double >
   class piecewise_constant_distribution;
 // 26.5.8.6.3, class template piecewise_linear_distribution
 template < class RealType = double >
   class piecewise_linear_distribution;
} // namespace std
```

## 26.5.3 Random number engine class templates

[rand.eng]

- Each type instantiated from a class template specified in this section 26.5.3 satisfies the requirements of a random number engine (26.5.1.4) type.
- Except where specified otherwise, the complexity of each function specified in this section 26.5.3 is constant.
- 3 Except where specified otherwise, no function described in this section 26.5.3 throws an exception.
- <sup>4</sup> 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.
- For every random number engine and for every random number engine adaptor X defined in this subclause (26.5.3) and in sub-clause 26.5.4:
  - if the constructor

```
template <class Sseq> explicit X(Sseq& q);
```

is called with a type Sseq that does not qualify as a seed sequence, then this constructor shall not participate in overload resolution:

— if the member function

§ 26.5.3

```
template <class Sseq> void seed(Sseq& q);
```

is called with a type Sseq that does not qualify as a seed sequence, then this function shall not participate in overload resolution.

The extent to which an implementation determines that a type cannot be a seed sequence is unspecified, except that as a minimum a type shall not qualify as a seed sequence if it is implicitly convertible to X::result\_type.

#### 26.5.3.1 Class template linear congruential engine

[rand.eng.lcong]

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  $GA(x_i) = x_{i+1}$ .

```
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 == Ou ? 1u: Ou };
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);
};
```

- <sup>2</sup> 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]
- $^3$  If the template parameter m is not 0, the following relations shall hold: a < m and c < m.
- <sup>4</sup> The textual representation consists of the value of  $x_i$ .

```
explicit linear_congruential_engine(result_type s = default_seed);
```

Effects: Constructs a linear\_congruential\_engine object. If  $c \mod m$  is 0 and s mod m is 0, sets the engine's state to 1, otherwise sets the engine's state to s mod m.

template<class Sseq> explicit linear\_congruential\_engine(Sseq& q);

§ 26.5.3.1 898

Effects: Constructs a linear\_congruential\_engine object. With  $k = \left\lceil \frac{\log_2 m}{32} \right\rceil$  and a an array (or equivalent) of length k+3, invokes q.generate(a+0, a+k+3) and then computes  $S = \left(\sum_{j=0}^{k-1} a_{j+3} \cdot 2^{32j}\right) \mod m$ . If  $c \mod m$  is 0 and S is 0, sets the engine's state to 1, else sets the engine's state to S.

#### 26.5.3.2 Class template mersenne\_twister\_engine

[rand.eng.mers]

- A mersenne\_twister\_engine random number engine<sup>275</sup> produces unsigned integer random numbers in the closed interval  $[0, 2^w 1]$ . The state  $x_i$  of a mersenne\_twister\_engine object x is of size n and consists of a sequence X of n values of the type delivered by x; all subscripts applied to X are to be taken modulo n.
- The transition algorithm employs a twisted generalized feedback shift register defined by shift values n and m, a twist value r, and a conditional xor-mask a. To improve the uniformity of the result, the bits of the raw shift register are additionally tempered (i.e., scrambled) according to a bit-scrambling matrix defined by values u, d, s, b, t, c, and  $\ell$ .

The state transition is performed as follows:

- a) Concatenate the upper w-r bits of  $X_{i-n}$  with the lower r bits of  $X_{i+1-n}$  to obtain an unsigned integer value Y.
- b) With  $\alpha = a \cdot (Y \text{ bit and } 1)$ , set  $X_i$  to  $X_{i+m-n}$  xor (Y rshift 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 \operatorname{xor} ((X_i \operatorname{rshift} u) \operatorname{bitand} d).
b) Let z_2 = z_1 \operatorname{xor} ((z_1 \operatorname{lshift}_w s) \operatorname{bitand} b).
c) Let z_3 = z_2 \operatorname{xor} ((z_2 \operatorname{lshift}_w t) \operatorname{bitand} c).
d) Let z_4 = z_3 \operatorname{xor}(z_3 \operatorname{rshift} \ell).
template < class UIntType, size_t w, size_t n, size_t m, size_t r,
             UIntType a, size_t u, UIntType d, size_t s,
             UIntType b, size_t t,
             UIntType c, size_t l, UIntType f>
 class mersenne_twister_engine
public:
 // types
 typedef UIntType result_type;
 // engine characteristics
 static constexpr size_t word_size = w;
 static constexpr size_t state_size = n;
 static constexpr size_t shift_size = m;
 static constexpr size_t mask_bits = r;
 static constexpr UIntType xor_mask = a;
 static constexpr size_t tempering_u = u;
 static constexpr UIntType tempering_d = d;
 static constexpr size_t tempering_s = s;
 static constexpr UIntType tempering_b = b;
```

275) 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.

§ 26.5.3.2

```
static constexpr size_t tempering_t = t;
static constexpr UIntType tempering_c = c;
static constexpr size_t tempering_l = l;
static constexpr UIntType initialization_multiplier = f;
static constexpr result_type min () { return 0; }
static constexpr result_type max() { return 2<sup>w</sup> - 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);
};
```

- 4 The following relations shall hold: 0 < m, m <= n, 2u < w, r <= w, u <= w, s <= w, t <= w, u <= numeric\_limits<UIntType>::digits, a <= (1u < w) 1u, b <= (1u < w) 1u, c <= (1u < w) 1u, d <= (1u < w) 1u, and f <= (1u < w) 1u.
- <sup>5</sup> The textual representation of  $x_i$  consists of the values of  $X_{i-n}, \ldots, X_{i-1}$ , in that order.

explicit mersenne\_twister\_engine(result\_type value = default\_seed);

Effects: Constructs a mersenne\_twister\_engine object. Sets  $X_{-n}$  to value mod  $2^w$ . Then, iteratively for  $i=1-n,\ldots,-1$ , sets  $X_i$  to

$$[f \cdot (X_{i-1} \operatorname{xor} (X_{i-1} \operatorname{rshift} (w-2))) + i \operatorname{mod} n] \operatorname{mod} 2^w.$$

<sup>7</sup> Complexity:  $\mathcal{O}(n)$ .

8

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  $n \cdot k$ , invokes q.generate  $(a+0, a+n \cdot k)$  and then, iteratively for  $i=-n,\ldots,-1$ , sets  $X_i$  to  $\left(\sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32j}\right)$  mod  $2^w$ . Finally, if the most significant w-r bits of  $X_{-n}$  are zero, and if each of the other resulting  $X_i$  is 0, changes  $X_{-n}$  to  $2^{w-1}$ .

## 26.5.3.3 Class template subtract\_with\_carry\_engine

[rand.eng.sub]

- <sup>1</sup> A subtract\_with\_carry\_engine random number engine produces unsigned integer random numbers.
- The state  $x_i$  of a subtract\_with\_carry\_engine object x is of size  $\mathcal{O}(r)$ , and consists of a sequence X of r integer values  $0 \le X_i < m = 2^w$ ; all subscripts applied to X are to be taken modulo r. The state  $x_i$  additionally consists of an integer c (known as the carry) whose value is either 0 or 1.
- 3 The state transition is performed as follows:
  - a) Let  $Y = X_{i-s} X_{i-r} c$ .
  - b) Set  $X_i$  to  $y = Y \mod m$ . Set c to 1 if Y < 0, otherwise set c to 0.

[Note: This algorithm corresponds to a modular linear function of the form  $\mathsf{TA}(\mathbf{x}_i) = (a \cdot \mathbf{x}_i) \bmod b$ , where b is of the form  $m^r - m^s + 1$  and a = b - (b-1)/m. — end note]

The generation algorithm is given by  $GA(x_i) = y$ , where y is the value produced as a result of advancing the engine's state as described above.

§ 26.5.3.3

```
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: Ou < s, s < r, O < w, and w <= numeric limits<UIntType>::digits.
   The textual representation consists of the values of X_{i-r}, \ldots, X_{i-1}, in that order, followed by c.
   explicit subtract_with_carry_engine(result_type value = default_seed);
7
         Effects: Constructs a subtract_with_carry_engine object. Sets the values of X_{-r}, \ldots, X_{-1}, in that
         order, as specified below. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.
         To set the values X_k, first construct e, a linear_congruential_engine object, as if by the following
         definition:
           linear_congruential_engine<result_type,</pre>
                                        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.
8
         Complexity: Exactly n \cdot \mathbf{r} invocations of \mathbf{e}.
   template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);
9
         Effects: Constructs a subtract_with_carry_engine object. With k = \lceil w/32 \rceil and a an array (or
         equivalent) of length r \cdot k, invokes q.generate (a+0, a+r \cdot k) and then, iteratively for i=-r,\ldots,-1,
         sets X_i to \left(\sum_{j=0}^{k-1} a_{k(i+r)+j} \cdot 2^{32j}\right) mod m. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.
   26.5.4
             Random number engine adaptor class templates
                                                                                                   [rand.adapt]
```

### 26.5.4.1 In general

[rand.adapt.general]

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.

§ 26.5.4.1 901

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.
- <sup>4</sup> 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.
- <sup>5</sup> Each template specified in this section 26.5.4 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

#### 26.5.4.2 Class template discard\_block\_engine

[rand.adapt.disc]

- A discard\_block\_engine random number engine adaptor produces random numbers selected from those produced by some base engine e. The state  $x_i$  of a discard\_block\_engine engine adaptor object x consists of the state  $e_i$  of its base engine e and an additional integer n. The size of the state is the size of e's state plus 1.
- The transition algorithm discards all but r > 0 values from each block of  $p \ge r$  values delivered by e. The state transition is performed as follows: If  $n \ge r$ , 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  $e_i$  to  $e_{i+1}$ .
- <sup>3</sup> 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 noexcept { return e; };
private:
             // exposition only
Engine e;
int n;
             // exposition only
};
```

§ 26.5.4.2

- The following relations shall hold: 0 < r and r <= p.
- <sup>5</sup> The textual representation consists of the textual representation of e followed by the value of n.
- <sup>6</sup> In addition to its behavior pursuant to section 26.5.1.5, each constructor that is not a copy constructor sets n to 0.

## 26.5.4.3 Class template independent\_bits\_engine

[rand.adapt.ibits]

- An independent\_bits\_engine random number engine adaptor combines random numbers that are produced by some base engine e, so as to produce random numbers with a specified number of bits w. The state  $\mathbf{x}_i$  of an independent\_bits\_engine engine adaptor object  $\mathbf{x}$  consists of the state  $\mathbf{e}_i$  of its base engine  $\mathbf{e}$ ; the size of the state is the size of e's state.
- <sup>2</sup> The transition and generation algorithms are described in terms of the following integral constants:
  - a) Let R = e.max() e.min() + 1 and  $m = \lfloor \log_2 R \rfloor$ .
  - b) With n as determined below, let  $w_0 = \lfloor w/n \rfloor$ ,  $n_0 = n w \mod n$ ,  $y_0 = 2^{w_0} \lfloor R/2^{w_0} \rfloor$ , and  $y_1 = 2^{w_0+1} \lfloor R/2^{w_0+1} \rfloor$ .
  - c) Let  $n = \lceil w/m \rceil$  if and only if the relation  $R y_0 \le \lfloor y_0/n \rfloor$  holds as a result. Otherwise let  $n = 1 + \lceil w/m \rceil$ .

[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  $y_1 + e.min()$ .
- <sup>4</sup> The generation algorithm uses the values produced while advancing the state as described above to yield a quantity S obtained as if by the following algorithm:

```
S = 0;
for (k = 0; k \neq n_0; k += 1) {
 do u = e() - e.min(); while (u \ge y_0);
 S = 2^{w_0} \cdot S + u \mod 2^{w_0};
for (k = n_0; k \neq n; k += 1) {
 do u = e() - e.min(); while (u \ge y_1);
 S = 2^{w_0+1} \cdot S + u \bmod 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);
```

§ 26.5.4.3 903

```
// generating functions
 result_type operator()();
 void discard(unsigned long long z);
 // property functions
 const Engine& base() const noexcept { return e; };
private:
Engine e;
             // exposition only
```

- The following relations shall hold: 0 < w and w <= numeric\_limits<result\_type>::digits.
- The textual representation consists of the textual representation of e.

### 26.5.4.4 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.

```
template<class Engine, size_t k>
class shuffle_order_engine
₹
public:
 typedef typename Engine::result_type result_type;
 // engine characteristics
 static constexpr size_t table_size = k;
 static constexpr result_type min() { return Engine::min(); }
 static constexpr result_type max() { return Engine::max(); }
 // constructors and seeding functions
 shuffle_order_engine();
 explicit shuffle_order_engine(const Engine& e);
 explicit shuffle_order_engine(Engine&& e);
 explicit shuffle_order_engine(result_type s);
 template<class Sseq> explicit shuffle_order_engine(Sseq& q);
 void seed();
 void seed(result_type s);
 template<class Sseq> void seed(Sseq& q);
 // generating functions
 result_type operator()();
 void discard(unsigned long long z);
 // property functions
 const Engine& base() const noexcept { return e; };
```

§ 26.5.4.4 904

4 The following relation shall hold: 0 < k.

1

- <sup>5</sup> The textual representation consists of the textual representation of **e**, followed by the **k** values of *V*, followed by the value of *Y*.
- In addition to its behavior pursuant to section 26.5.1.5, each constructor that is not a copy constructor initializes  $V[0], \ldots, V[k-1]$  and Y, in that order, with values returned by successive invocations of e().

## 26.5.5 Engines and engine adaptors with predefined parameters [rand.predef]

```
typedef linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647>
    minstd_rand0;
```

Required behavior: The 10000 <sup>th</sup> consecutive invocation of a default-constructed object of type minstd\_rand0 shall produce the value 1043618065.

```
typedef linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647>
    minstd_rand;
```

2 Required behavior: The 10000 <sup>th</sup> consecutive invocation of a default-constructed object of type minstd\_rand shall produce the value 399268537.

Required behavior: The 10000 th 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,
    0x55555555555555555,17,
    0x71d67fffeda60000,37,
    0xfff7eee0000000000,43,
    6364136223846793005>
    mt19937_64;
```

4 Required behavior: The 10000 th 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 10000 th consecutive invocation of a default-constructed object of type ranlux24\_base shall produce the value 7937952.

```
typedef subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>
    ranlux48_base;
```

§ 26.5.5 905

Required behavior: The 10000 <sup>th</sup> 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;
```

<sup>7</sup> Required behavior: The 10000 <sup>th</sup> 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 10000 th consecutive invocation of a default-constructed object of type ranlux48 shall produce the value 249142670248501.

```
typedef shuffle_order_engine<minstd_rand0,256>
    knuth_b;
```

Required behavior: The 10000 <sup>th</sup> consecutive invocation of a default-constructed object of type knuth\_b shall produce the value 1112339016.

```
typedef implementation-defined
    default_random_engine;
```

8

9

10

Remark: The choice of engine type named by this typedef is implementation-defined. [Note: The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use. Because different implementations may select different underlying engine types, code that uses this typedef need not generate identical sequences across implementations. — end note]

### 26.5.6 Class random\_device

[rand.device]

- <sup>1</sup> A random\_device uniform random number generator produces non-deterministic random numbers.
- <sup>2</sup> If implementation limitations prevent generating non-deterministic random numbers, the implementation may employ a random number engine.

```
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()();
```

§ 26.5.6 906

```
// property functions
   double entropy() const noexcept;
   // no copy functions
   random_device(const random_device& ) = delete;
   void operator=(const random_device& ) = delete;
  };
explicit random_device(const string& token = implementation-defined);
      Effects: Constructs a random_device non-deterministic uniform random number generator object.
      The semantics and default value of the token parameter are implementation-defined.<sup>276</sup>
      Throws: A value of an implementation-defined type derived from exception if the random_device
      could not be initialized.
double entropy() const noexcept;
      Returns: If the implementation employs a random number engine, returns 0.0. Otherwise, returns
      an entropy estimate<sup>277</sup> for the random numbers returned by operator(), in the range min() to
      \log_2(\max()+1).
result_type operator()();
      Returns: A non-deterministic random value, uniformly distributed between min() and max(), inclusive.
      It is implementation-defined how these values are generated.
      Throws: A value of an implementation-defined type derived from exception if a random number could
      not be obtained.
26.5.7 Utilities
                                                                                            [rand.util]
26.5.7.1 Class seed_seq
                                                                                    [rand.util.seedseq]
No function described in this section 26.5.7.1 throws an exception.
  class seed_seq
```

3

4

5

6

7

```
{
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 Random AccessIterator >
   void generate(RandomAccessIterator begin, RandomAccessIterator end);
```

276) The parameter is intended to allow an implementation to differentiate between different sources of randomness. 277) If a device has n states whose respective probabilities are  $P_0, \ldots, P_{n-1}$ , the device entropy S is defined as  $S = -\sum_{i=0}^{n-1} P_i \cdot \log P_i.$ 

§ 26.5.7.1 907

```
// 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
   seed_seq();
        Effects: Constructs a seed_seq object as if by default-constructing its member v.
   template<class T>
    seed_seq(initializer_list<T> il);
3
        Requires: T shall be an integer type.
4
        Effects: Same as seed_seq(il.begin(), il.end()).
   template < class InputIterator>
     seed_seq(InputIterator begin, InputIterator end);
5
        Requires: InputIterator shall satisfy the requirements of an input iterator (Table 107) type. More-
        over, iterator_traits<InputIterator>::value_type shall denote an integer type.
6
        Effects: Constructs a seed_seq object by the following algorithm:
          for( InputIterator s = begin; s != end; ++s)
           v.push_back((*s)mod2^{32});
   template < class Random AccessIterator >
     void generate(RandomAccessIterator begin, RandomAccessIterator end);
7
        Requires: RandomAccessIterator shall meet the requirements of a mutable random access iterator
        (Table 111) type. Moreover, iterator_traits<RandomAccessIterator>::value_type shall denote
        an unsigned integer type capable of accommodating 32-bit quantities.
8
        Effects: Does nothing if begin == end. Otherwise, with s = v.size() and n = end - begin, fills
```

the supplied range [begin, end) according to the following algorithm in which each operation is to be carried out modulo  $2^{32}$ , each indexing operator applied to begin is to be taken modulo n, and T(x) is defined as  $x \times (x \text{ rshift } 27)$ :

a) By way of initialization, set each element of the range to the value 0x8b8b8b8b. Additionally, for use in subsequent steps, let p = (n - t)/2 and let q = p + t, where

```
t = (n > 623)? 11: (n > 68)? 7: (n > 39)? 5: (n > 7)? 3: (n - 1)/2;
```

§ 26.5.7.1 908

b) With m as the larger of s+1 and n, transform the elements of the range: iteratively for  $k=0,\ldots,m-1$ , calculate values

$$\begin{array}{lll} r_1 & = & 1664525 \cdot {\tt T} \left( {\tt begin} [k] \, {\tt xor} \, {\tt begin} [k+p] \, {\tt xor} \, {\tt begin} [k-1] \right) \\ r_2 & = & r_1 + \left\{ \begin{array}{ll} s & , \ k=0 \\ k \bmod n + {\tt v} [k-1] & , \ 0 < k \leq s \\ k \bmod n & , \ s < k \end{array} \right. \\ \end{array}$$

and, in order, increment begin [k+p] by  $r_1$ , increment begin [k+q] by  $r_2$ , and set begin [k] to  $r_2$ .

c) Transform the elements of the range again, beginning where the previous step ended: iteratively for k = m, ..., m+n-1, calculate values

$$r_3 = 1566083941 \cdot T(\text{begin}[k] + \text{begin}[k + p] + \text{begin}[k - 1])$$
  
 $r_4 = r_3 - (k \mod n)$ 

and, in order, update begin[k+p] by xoring it with  $r_3$ , update begin[k+q] by xoring it with  $r_4$ , and set begin[k] to  $r_4$ .

size\_t size() const;

- 9 Returns: The number of 32-bit units that would be returned by a call to param().
- 10 Complexity: constant time.

template<class OutputIterator>
 void param(OutputIterator dest) const;

- Requires: OutputIterator shall satisfy the requirements of an output iterator (Table 108) type. Moreover, the expression \*dest = rt shall be valid for a value rt of type result\_type.
- $^{12}$  Effects: Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:

copy(v.begin(), v.end(), dest);

## 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  $t_j$ ,  $0 \le t_j < 1$ , 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]

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  $b^{278}$  is the lesser of numeric\_limits<RealType>::digits and bits, and R is the value of g.max() - g.min() + 1.

278) b is introduced to avoid any attempt to produce more bits of randomness than can be held in RealType.

§ 26.5.7.2 909

4 Effects: Invokes g() k times to obtain values  $g_0, \ldots, g_{k-1}$ , respectively. Calculates a quantity

$$S = \sum_{i=0}^{k-1} (g_i - \mathsf{g.min()}) \cdot R^i$$

using arithmetic of type RealType.

- 5 Returns:  $S/R^k$ .
- 6 Throws: What and when g throws.

#### 26.5.8 Random number distribution class templates

[rand.dist]

#### 26.5.8.1 In general

[rand.dist.general]

- <sup>1</sup> 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.
- 3 The algorithms for producing each of the specified distributions are implementation-defined.
- <sup>4</sup> The value of each probability density function p(z) and of each discrete probability function  $P(z_i)$  specified in this section is 0 everywhere outside its stated domain.

#### 26.5.8.2 Uniform distributions

[rand.dist.uni]

#### 26.5.8.2.1 Class template uniform int distribution

[rand.dist.uni.int]

A uniform\_int\_distribution random number distribution produces random integers  $i, a \leq i \leq b$ , distributed according to the constant discrete probability function

$$P(i | a, b) = 1/(b - a + 1)$$
.

```
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;
```

§ 26.5.8.2.1

```
void param(const param_type& parm);
      result_type min() const;
     result_type max() const;
     };
   explicit uniform_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());
2
        Requires: a \leq b.
3
        Effects: Constructs a uniform_int_distribution object; a and b correspond to the respective pa-
        rameters of the distribution.
   result_type a() const;
4
        Returns: The value of the a parameter with which the object was constructed.
   result_type b() const;
5
        Returns: The value of the b parameter with which the object was constructed.
```

## 26.5.8.2.2 Class template uniform\_real\_distribution

[rand.dist.uni.real]

A uniform\_real\_distribution random number distribution produces random numbers  $x, a \leq x < b$ , distributed according to the constant probability density function

$$p(x | a, b) = 1/(b-a)$$
.

```
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;
};
```

§ 26.5.8.2.2 911

```
explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);

Requires: a \leq b \text{ and b - a \leq numeric_limits < RealType >::max().}

Effects: Constructs a uniform_real_distribution object; a and b correspond
```

Effects: Constructs a uniform\_real\_distribution object; a and b correspond to the respective parameters of the distribution.

result\_type a() const;

4 Returns: The value of the a parameter with which the object was constructed.

result\_type b() const;

2

3

5 Returns: The value of the b parameter with which the object was constructed.

#### 26.5.8.3 Bernoulli distributions

[rand.dist.bern]

#### 26.5.8.3.1 Class bernoulli distribution

[rand.dist.bern.bernoulli]

A bernoulli\_distribution random number distribution produces bool values b distributed according to the discrete probability function

$$P(b \,|\, p) = \left\{ egin{array}{ll} p & ext{if} & b = ext{true} \\ 1-p & ext{if} & b = ext{false} \end{array} 
ight. \,.$$

```
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 \le p \le 1.
     Effects: Constructs a bernoulli_distribution object; p corresponds to the parameter of the distri-
     bution.
```

§ 26.5.8.3.1 912

```
double p() const;
```

2

3

5

4 Returns: The value of the p parameter with which the object was constructed.

#### 26.5.8.3.2 Class template binomial\_distribution

[rand.dist.bern.bin]

A binomial\_distribution random number distribution produces integer values  $i \geq 0$  distributed according to the discrete probability function

$$P(i \mid t, p) = {t \choose i} \cdot p^i \cdot (1 - p)^{t - i} .$$

```
template<class IntType = int>
   class binomial_distribution
  public:
   // types
   typedef IntType result_type;
   typedef unspecified param_type;
   // constructors and reset functions
   explicit binomial_distribution(IntType t = 1, double p = 0.5);
   explicit binomial_distribution(const param_type& parm);
   void reset();
   // generating functions
   template<class URNG>
     result_type operator()(URNG& g);
   template<class URNG>
     result_type operator()(URNG& g, const param_type& parm);
   // property functions
   IntType t() const;
   double p() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
  };
explicit binomial_distribution(IntType t = 1, double p = 0.5);
     Requires: 0 \le p \le 1 and 0 \le t.
     Effects: Constructs a binomial_distribution object; t and p correspond to the respective parameters
     of the distribution.
IntType t() const;
     Returns: The value of the t parameter with which the object was constructed.
double p() const;
     Returns: The value of the p parameter with which the object was constructed.
```

§ 26.5.8.3.2 913

#### 26.5.8.3.3 Class template geometric\_distribution

[rand.dist.bern.geo]

A geometric\_distribution random number distribution produces integer values  $i \geq 0$  distributed according to the discrete probability function

$$P(i \mid p) = p \cdot (1 - p)^i.$$

```
template<class IntType = int>
  class geometric_distribution
  {
 public:
  // types
  typedef IntType result_type;
   typedef unspecified param_type;
   // constructors and reset functions
   explicit geometric_distribution(double p = 0.5);
   explicit geometric_distribution(const param_type& parm);
   void reset();
   // generating functions
   template<class 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 geometric_distribution(double p = 0.5);
     Requires: 0 .
     Effects: Constructs a geometric_distribution object; p corresponds to the parameter of the distri-
double p() const;
```

4 Returns: The value of the p parameter with which the object was constructed.

#### 26.5.8.3.4 Class template negative\_binomial\_distribution [rand.dist.bern.negbin]

A negative\_binomial\_distribution random number distribution produces random integers  $i \geq 0$  distributed according to the discrete probability function

$$P(i \mid k, p) = {k+i-1 \choose i} \cdot p^k \cdot (1-p)^i.$$

```
template<class IntType = int>
  class negative_binomial_distribution
{
```

2

3

§ 26.5.8.3.4 914

```
public:
      // types
      typedef IntType result_type;
      typedef unspecified param_type;
      // constructor and reset functions
      explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);
      explicit negative_binomial_distribution(const param_type& parm);
      void reset();
      // generating functions
      template<class URNG>
        result_type operator()(URNG& g);
      template<class URNG>
        result_type operator()(URNG& g, const param_type& parm);
      // property functions
      IntType k() const;
      double p() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
      result_type max() const;
     };
   explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);
2
        Requires: 0  and <math>0 < k.
3
        Effects: Constructs a negative_binomial_distribution object; k and p correspond to the respective
        parameters of the distribution.
   IntType k() const;
        Returns: The value of the k parameter with which the object was constructed.
   double p() const;
        Returns: The value of the p parameter with which the object was constructed.
```

#### 26.5.8.4 Poisson distributions

[rand.dist.pois]

## 26.5.8.4.1 Class template poisson\_distribution

[rand.dist.pois.poisson]

A poisson\_distribution random number distribution produces integer values  $i \ge 0$  distributed according to the discrete probability function

$$P(i \mid \mu) = \frac{e^{-\mu}\mu^i}{i!} .$$

The distribution parameter  $\mu$  is also known as this distribution's mean.

```
template<class IntType = int>
  class poisson_distribution
{
  public:
    // types
```

§ 26.5.8.4.1

```
typedef IntType result_type;
      typedef unspecified param_type;
      // constructors and reset functions
      explicit poisson_distribution(double mean = 1.0);
      explicit poisson_distribution(const param_type& parm);
      void reset();
      // generating functions
      template<class URNG>
        result_type operator()(URNG& g);
      template<class URNG>
        result_type operator()(URNG& g, const param_type& parm);
      // property functions
      double mean() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
     result_type max() const;
     };
   explicit poisson_distribution(double mean = 1.0);
2
        Requires: 0 < mean.
3
        Effects: Constructs a poisson_distribution object; mean corresponds to the parameter of the dis-
        tribution.
   double mean() const;
```

#### 26.5.8.4.2 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

Returns: The value of the mean parameter with which the object was constructed.

$$p(x \mid \lambda) = \lambda e^{-\lambda x}$$
.

```
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);
```

§ 26.5.8.4.2

```
template<class URNG>
        result_type operator()(URNG& g, const param_type& parm);
      // property functions
      RealType lambda() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
     result_type max() const;
     };
   explicit exponential_distribution(RealType lambda = 1.0);
2
        Requires: 0 < lambda.
3
        Effects: Constructs a exponential_distribution object; lambda corresponds to the parameter of the
        distribution.
   RealType lambda() const;
        Returns: The value of the lambda parameter with which the object was constructed.
```

### 26.5.8.4.3 Class template gamma\_distribution

[rand.dist.pois.gamma]

A gamma\_distribution random number distribution produces random numbers x > 0 distributed according to the probability density function

$$p(x \mid \alpha, \beta) = \frac{e^{-x/\beta}}{\beta^{\alpha} \cdot \Gamma(\alpha)} \cdot x^{\alpha - 1} .$$

```
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;
};
```

§ 26.5.8.4.3 917

```
explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0); Requires: 0 < alpha and 0 < beta.
```

3 Effects: Constructs a gamma\_distribution object; alpha and beta correspond to the parameters of the distribution.

RealType alpha() const;

2

2

3

Returns: The value of the alpha parameter with which the object was constructed.

RealType beta() const;

5 Returns: The value of the beta parameter with which the object was constructed.

#### 26.5.8.4.4 Class template weibull\_distribution

[rand.dist.pois.weibull]

A weibull\_distribution random number distribution produces random numbers  $x \geq 0$  distributed according to the probability density function

$$p(x \mid a, b) = \frac{a}{b} \cdot \left(\frac{x}{b}\right)^{a-1} \cdot \exp\left(-\left(\frac{x}{b}\right)^{a}\right) .$$

```
template < class RealType = double >
   class weibull_distribution
  public:
   // types
   typedef RealType result_type;
   typedef unspecified param_type;
   // constructor and reset functions
   explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0)
   explicit weibull_distribution(const param_type& parm);
   void reset();
   // generating functions
   template<class URNG>
     result_type operator()(URNG& g);
   template<class URNG>
     result_type operator()(URNG& g, const param_type& parm);
   // property functions
   RealType a() const;
   RealType b() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
  };
explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);
     Requires: 0 < a and 0 < b.
     Effects: Constructs a weibull_distribution object; a and b correspond to the respective parameters
     of the distribution.
```

§ 26.5.8.4.4 918

```
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.

## 26.5.8.4.5 Class template extreme\_value\_distribution

[rand.dist.pois.extreme]

An extreme\_value\_distribution random number distribution produces random numbers x distributed according to the probability density function<sup>279</sup>

$$p(x \mid a, b) = \frac{1}{b} \cdot \exp\left(\frac{a - x}{b} - \exp\left(\frac{a - x}{b}\right)\right).$$

```
template < class RealType = double >
   class extreme_value_distribution
  public:
   // types
   typedef RealType result_type;
   typedef unspecified param_type;
   // constructor and reset functions
   explicit extreme_value_distribution(RealType a = 0.0, RealType b = 1.0);
   explicit extreme_value_distribution(const param_type& parm);
   void reset();
   // generating functions
   template<class URNG>
     result_type operator()(URNG& g);
   template<class URNG>
     result_type operator()(URNG& g, const param_type& parm);
   // property functions
   RealType a() const;
   RealType b() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
  };
explicit extreme_value_distribution(RealType a = 0.0, RealType b = 1.0);
     Requires: 0 < b.
```

Effects: Constructs an extreme\_value\_distribution object; a and b correspond to the respective parameters of the distribution.

RealType a() const;

2

§ 26.5.8.4.5 919

<sup>279)</sup> 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.

4 Returns: The value of the a parameter with which the object was constructed.

RealType b() const;

Returns: The value of the b parameter with which the object was constructed.

#### 26.5.8.5 Normal distributions

[rand.dist.norm]

#### 26.5.8.5.1 Class template normal\_distribution

[rand.dist.norm.normal]

A normal\_distribution random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

The distribution parameters  $\mu$  and  $\sigma$  are also known as this distribution's mean and standard deviation.

```
template < class RealType = double >
  class normal_distribution
  {
 public:
  // types
  typedef RealType result_type;
  typedef unspecified param_type;
   // constructors and reset functions
  explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);
  explicit normal_distribution(const param_type& parm);
  void reset();
   // generating functions
  template<class URNG>
    result_type operator()(URNG& g);
  template<class URNG>
    result_type operator()(URNG& g, const param_type& parm);
   // property functions
  RealType mean() const;
  RealType stddev() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);
     Requires: 0 < stddev.
     Effects: Constructs a normal_distribution object; mean and stddev correspond to the respective
     parameters of the distribution.
```

RealType mean() const;

2

3

4 Returns: The value of the mean parameter with which the object was constructed.

§ 26.5.8.5.1 920

RealType stddev() const;

2

3

4

5

Returns: The value of the stddev parameter with which the object was constructed.

#### 26.5.8.5.2 Class template lognormal\_distribution

[rand.dist.norm.lognormal]

A lognormal\_distribution random number distribution produces random numbers x > 0 distributed according to the probability density function

$$p(x \mid m, s) = \frac{1}{sx\sqrt{2\pi}} \cdot \exp\left(-\frac{(\ln x - m)^2}{2s^2}\right).$$

```
template < class RealType = double >
   class lognormal_distribution
  public:
   // types
   typedef RealType result_type;
   typedef unspecified param_type;
   // constructor and reset functions
   explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
   explicit lognormal_distribution(const param_type& parm);
   void reset();
   // generating functions
   template<class URNG>
     result_type operator()(URNG& g);
   template<class URNG>
     result_type operator()(URNG& g, const param_type& parm);
   // property functions
   RealType m() const;
   RealType s() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
  };
explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
     Requires: 0 < s.
     Effects: Constructs a lognormal distribution object; m and s correspond to the respective param-
     eters of the distribution.
RealType m() const;
     Returns: The value of the m parameter with which the object was constructed.
RealType s() const;
     Returns: The value of the s parameter with which the object was constructed.
```

§ 26.5.8.5.2 921

#### 26.5.8.5.3 Class template chi\_squared\_distribution

[rand.dist.norm.chisq]

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);
     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.

### 26.5.8.5.4 Class template cauchy\_distribution

3

[rand.dist.norm.cauchy]

A cauchy\_distribution random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid a, b) = \left(\pi b \left(1 + \left(\frac{x - a}{b}\right)^2\right)\right)^{-1}.$$

§ 26.5.8.5.4 922

```
template < class RealType = double >
      class cauchy_distribution
     {
     public:
      // types
      typedef RealType result_type;
      typedef unspecified param_type;
      // constructor and reset functions
      explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
      explicit cauchy_distribution(const param_type& parm);
      void reset();
      // generating functions
      template<class URNG>
         result_type operator()(URNG& g);
      template<class URNG>
        result_type operator()(URNG& g, const param_type& parm);
      // property functions
      RealType a() const;
      RealType b() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
      result_type max() const;
     };
   explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
2
         Requires: 0 < b.
3
         Effects: Constructs a cauchy_distribution object; a and b correspond to the respective parameters
         of the distribution.
   RealType a() const;
4
         Returns: The value of the a parameter with which the object was constructed.
   RealType b() const;
         Returns: The value of the b parameter with which the object was constructed.
   26.5.8.5.5 Class template fisher_f_distribution
                                                                                               [rand.dist.norm.f]
  A fisher_f_distribution random number distribution produces random numbers x \geq 0 distributed ac-
   cording to the probability density function
                     p(x \, | \, m,n) = \frac{\Gamma \left( (m+n)/2 \right)}{\Gamma (m/2) \, \Gamma (n/2)} \cdot \left( \frac{m}{n} \right)^{m/2} \cdot x^{(m/2)-1} \cdot \left( 1 + \frac{mx}{n} \right)^{-(m+n)/2} \, .
     template < class RealType = double >
      class fisher_f_distribution
     public:
   § 26.5.8.5.5
                                                                                                                923
```

```
// types
      typedef RealType result_type;
      typedef unspecified param_type;
      // constructor and reset functions
      explicit fisher_f_distribution(RealType m = 1, RealType n = 1);
      explicit fisher_f_distribution(const param_type& parm);
      void reset();
      // generating functions
      template<class URNG>
        result_type operator()(URNG& g);
      template<class URNG>
        result_type operator()(URNG& g, const param_type& parm);
      // property functions
      RealType m() const;
      RealType n() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
      result_type max() const;
     };
   explicit fisher_f_distribution(RealType m = 1, RealType n = 1);
2
        Requires: 0 < m and 0 < n.
3
        Effects: Constructs a fisher_f_distribution object; m and n correspond to the respective parameters
        of the distribution.
   RealType m() const;
4
         Returns: The value of the m parameter with which the object was constructed.
   RealType n() const;
5
        Returns: The value of the n parameter with which the object was constructed.
```

### 26.5.8.5.6 Class template student\_t\_distribution

[rand.dist.norm.t]

A student\_t\_distribution random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid n) = \frac{1}{\sqrt{n\pi}} \cdot \frac{\Gamma((n+1)/2)}{\Gamma(n/2)} \cdot \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2} .$$

```
template<class RealType = double>
  class student_t_distribution
{
  public:
    // types
    typedef RealType result_type;
  typedef unspecified param_type;
```

§ 26.5.8.5.6 924

```
// 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;
     };
   explicit student_t_distribution(RealType n = 1);
2
        Requires: 0 < n.
3
        Effects: Constructs a student_t_distribution object; n corresponds to the parameter of the distri-
        bution.
   RealType n() const;
```

4 Returns: The value of the n parameter with which the object was constructed.

# 26.5.8.6 Sampling distributions

[rand.dist.samp]

# 26.5.8.6.1 Class template discrete\_distribution

[rand.dist.samp.discrete]

A discrete\_distribution random number distribution produces random integers i,  $0 \le i < n$ , distributed according to the discrete probability function

$$P(i | p_0, \dots, p_{n-1}) = p_i$$
.

Unless specified otherwise, the distribution parameters are calculated as:  $p_k = w_k/S$  for k = 0, ..., n-1, in which the values  $w_k$ , commonly known as the weights, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold:  $0 < S = w_0 + \cdots + w_{n-1}$ .

```
template<class IntType = int>
  class discrete_distribution
{
public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    discrete_distribution();
    template<class InputIterator>
        discrete_distribution(InputIterator firstW, InputIterator lastW);
    discrete_distribution(initializer_list<double> wl);
```

§ 26.5.8.6.1

```
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;
      };
   discrete_distribution();
3
         Effects: Constructs a discrete_distribution object with n=1 and p_0=1. [Note: Such an object
         will always deliver the value 0. -end note
   template < class InputIterator>
      discrete_distribution(InputIterator firstW, InputIterator lastW);
         Requires: InputIterator shall satisfy the requirements of an input iterator (Table 107) type. More-
         over, iterator_traits<InputIterator>::value_type shall denote a type that is convertible to
         double. If firstW == lastW, let n = 1 and w_0 = 1. Otherwise, [firstW, lastW) shall form a se-
         quence w of length n > 0.
5
         Effects: Constructs a discrete_distribution object with probabilities given by the formula above.
   discrete_distribution(initializer_list<double> wl);
         Effects: Same as discrete_distribution(wl.begin(), wl.end()).
   template < class UnaryOperation>
      discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);
7
         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 < \delta = (xmax - xmin)/n shall
8
         Effects: Constructs a discrete_distribution object with probabilities given by the formula above,
         using the following values: If nw = 0, let w_0 = 1. Otherwise, let w_k = fw(xmin + k \cdot \delta + \delta/2) for
9
         Complexity: The number of invocations of fw shall not exceed n.
   vector<double> probabilities() const;
10
         Returns: A vector double whose size member returns n and whose operator member returns
         p_k when invoked with argument k for k = 0, \dots, n-1.
```

926

§ 26.5.8.6.1

# 26.5.8.6.2 Class template piecewise\_constant\_distribution

[rand.dist.samp.pconst]

A piecewise\_constant\_distribution random number distribution produces random numbers  $x, b_0 \le x < b_n$ , uniformly distributed over each subinterval  $[b_i, b_{i+1})$  according to the probability density function

$$p(x | b_0, \dots, b_n, \rho_0, \dots, \rho_{n-1}) = \rho_i$$
, for  $b_i \le x < b_{i+1}$ .

The n+1 distribution parameters  $b_i$ , also known as this distribution's *interval boundaries*, shall satisfy the relation  $b_i < b_{i+1}$  for i = 0, ..., n-1. Unless specified otherwise, the remaining n distribution parameters are calculated as:

$$\rho_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \text{ for } k = 0, \dots, n-1,$$

in which the values  $w_k$ , commonly known as the weights, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold:  $0 < S = w_0 + \cdots + w_{n-1}$ .

```
template < class RealType = double >
  class piecewise_constant_distribution
  {
  public:
  // types
   typedef RealType result_type;
   typedef unspecified param_type;
   // constructor and reset functions
   piecewise_constant_distribution();
   template < class InputIteratorB, class InputIteratorW>
     piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                      InputIteratorW firstW);
   template < class UnaryOperation>
     piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);
   template < class UnaryOperation>
     piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
   explicit piecewise_constant_distribution(const param_type& parm);
   void reset();
   // generating functions
   template<class URNG>
     result_type operator()(URNG& g);
   template<class URNG>
     result_type operator()(URNG& g, const param_type& parm);
   // property functions
   vector<result_type> intervals() const;
   vector<result_type> densities() const;
  param_type param() const;
   void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
  };
piecewise_constant_distribution();
     Effects: Constructs a piecewise_constant_distribution object with n=1, \rho_0=1, b_0=0, and
     b_1 = 1.
```

§ 26.5.8.6.2 927

3

Requires: InputIteratorB and InputIteratorW shall each satisfy the requirements of an input iterator (Table 107) type. Moreover, iterator\_traits<InputIteratorB>::value\_type and iterator\_traits<InputIteratorW>::value\_type shall each denote a type that is convertible to double. If firstB == lastB or ++firstB == lastB, let n=1,  $w_0=1$ ,  $b_0=0$ , and  $b_1=1$ . Otherwise, [firstB,lastB) shall form a sequence b of length n+1, the length of the sequence w starting from firstW shall be at least n, and any  $w_k$  for  $k \ge n$  shall be ignored by the distribution.

5 Effects: Constructs a piecewise\_constant\_distribution object with parameters as specified above.

```
template<class UnaryOperation>
piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);
```

- Requires: Each instance of type UnaryOperation shall be a function object (20.8) 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\_constant\_distribution object with parameters taken or calculated from the following values: If bl.size() < 2, let n = 1,  $w_0 = 1$ ,  $b_0 = 0$ , and  $b_1 = 1$ . Otherwise, let [bl.begin(), bl.end()) form a sequence  $b_0, \ldots, b_n$ , and let  $w_k = fw((b_{k+1} + b_k)/2)$  for  $k = 0, \ldots, n-1$ .
- 8 Complexity: The number of invocations of fw shall not exceed n.

```
template<class UnaryOperation>
piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
```

- Requires: Each instance of type UnaryOperation shall be a function object (20.8) 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  $0 < \delta = (xmax xmin)/n$  shall hold.
- Effects: Constructs a piecewise\_constant\_distribution object with parameters taken or calculated from the following values: Let  $b_k = \min + k \cdot \delta$  for  $k = 0, \dots, n$ , and  $w_k = \text{fw}(b_k + \delta/2)$  for  $k = 0, \dots, n-1$ .
- 11 Complexity: The number of invocations of fw shall not exceed n.

```
vector<result_type> intervals() const;
```

Returns: A vector<result\_type> whose size member returns n+1 and whose operator[] member returns  $b_k$  when invoked with argument k for  $k=0,\ldots,n$ .

```
vector<result_type> densities() const;
```

*Returns:* A vector<result\_type> whose size member returns n and whose operator[] member returns  $\rho_k$  when invoked with argument k for k = 0, ..., n-1.

§ 26.5.8.6.2 928

#### 26.5.8.6.3 Class template piecewise\_linear\_distribution

### [rand.dist.samp.plinear]

A piecewise\_linear\_distribution random number distribution produces random numbers  $x, b_0 \le x < b_n$ , distributed over each subinterval  $[b_i, b_{i+1})$  according to the probability density function

$$p(x | b_0, \dots, b_n, \rho_0, \dots, \rho_n) = \rho_i \cdot \frac{b_{i+1} - x}{b_{i+1} - b_i} + \rho_{i+1} \cdot \frac{x - b_i}{b_{i+1} - b_i}, \text{ for } b_i \le x < b_{i+1}.$$

The n+1 distribution parameters  $b_i$ , also known as this distribution's interval boundaries, shall satisfy the relation  $b_i < b_{i+1}$  for i = 0, ..., n-1. Unless specified otherwise, the remaining n+1 distribution parameters are calculated as  $\rho_k = w_k/S$  for k = 0, ..., n, in which the values  $w_k$ , commonly known as the weights at boundaries, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold:

$$0 < S = \frac{1}{2} \cdot \sum_{k=0}^{n-1} (w_k + w_{k+1}) \cdot (b_{k+1} - b_k).$$

```
template < class RealType = double >
  class piecewise_linear_distribution
 public:
   // types
   typedef RealType result_type;
   typedef unspecified param_type;
   // constructor and reset functions
   piecewise_linear_distribution();
   template < class InputIteratorB, class InputIteratorW>
     piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                    InputIteratorW firstW);
   template < class UnaryOperation >
     piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);
   template < class UnaryOperation >
     piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
   explicit piecewise_linear_distribution(const param_type& parm);
   void reset();
   // generating functions
   template<class URNG>
     result_type operator()(URNG& g);
   template<class URNG>
     result_type operator()(URNG& g, const param_type& parm);
   // property functions
   vector<result_type> intervals() const;
   vector<result_type> densities() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
  result_type max() const;
  };
piecewise_linear_distribution();
     Effects: Constructs a piecewise_linear_distribution object with n=1, \rho_0=\rho_1=1, b_0=0, and
     b_1 = 1.
```

§ 26.5.8.6.3 929

Requires: InputIteratorB and InputIteratorW shall each satisfy the requirements of an input iterator (Table 107) type. Moreover, iterator\_traits<InputIteratorB>::value\_type and iterator\_traits<InputIteratorW>::value\_type shall each denote a type that is convertible to double. If firstB == lastB or ++firstB == lastB, let n=1,  $\rho_0=\rho_1=1$ ,  $b_0=0$ , and  $b_1=1$ . Otherwise, [firstB,lastB) shall form a sequence b of length n+1, the length of the sequence w starting from firstW shall be at least n+1, and any  $w_k$  for  $k \ge n+1$  shall be ignored by the distribution.

5 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,  $\rho_0 = \rho_1 = 1$ ,  $b_0 = 0$ , and  $b_1 = 1$ . Otherwise, let [bl.begin(),bl.end()) form a sequence  $b_0, \ldots, b_n$ , and let  $w_k = fw(b_k)$  for  $k = 0, \ldots, n$ .
- 8 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  $0 < \delta = (xmax xmin)/n$  shall hold.
- Effects: Constructs a piecewise\_linear\_distribution object with parameters taken or calculated from the following values: Let  $b_k = \min + k \cdot \delta$  for k = 0, ..., n, and  $w_k = \mathsf{fw}(b_k)$  for k = 0, ..., n.
- 11 Complexity: The number of invocations of fw shall not exceed n+1.

```
vector<result_type> intervals() const;
```

Returns: A vector<result\_type> whose size member returns n+1 and whose operator[] member returns  $b_k$  when invoked with argument k for  $k=0,\ldots,n$ .

```
vector<result_type> densities() const;
```

*Returns:* A vector<result\_type> whose size member returns n and whose operator[] member returns  $\rho_k$  when invoked with argument k for k = 0, ..., n.

§ 26.5.8.6.3 930

# 26.6 Numeric arrays

#### 26.6.1 Header <valarray> synopsis

[numarray] [valarray.syn]

```
#include <initializer_list>
namespace std {
  template<class T> class valarray;
                                              // An array of type T
  class slice;
                                              // a BLAS-like slice out of an array
  template<class T> class slice_array;
  class gslice;
                                              // a generalized slice out of an array
  template<class T> class gslice_array;
                                              // a masked array
  template<class T> class mask_array;
  template<class T> class indirect_array;
                                              // an indirected array
  template<class T> void swap(valarray<T>&, valarray<T>&) noexcept;
  template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator* (const valarray<T>&, const T&);
  template<class T> valarray<T> operator* (const T&, const valarray<T>&);
  template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator/ (const valarray<T>&, const T&);
  template<class T> valarray<T> operator/ (const T&, const valarray<T>&);
  template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator% (const valarray<T>&, const T&);
  template<class T> valarray<T> operator% (const T&, const valarray<T>&);
  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&);
  \label{template} $$ $$ $$ 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&);
```

§ 26.6.1 931

```
template<class T> valarray<T> operator>>(const T&, const valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&, const valarray<T>\);
template<class T> valarray<bool> operator&&(const valarray<T>&, const T&);
template<class T> valarray<bool> operator&&(const T&, const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator | (const valarray<T>&, const T&);
template<class T> valarray<bool> operator||(const T&, const valarray<T>&);
template<class T>
 valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator==(const valarray<T>&, const T&);
template<class T> valarray<bool> operator==(const T&, const valarray<T>&);
template<class T>
  valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator!=(const T&, const valarray<T>&);
template<class T>
 valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&, const T&);
template<class T> valarray<bool> operator< (const T&, const valarray<T>&);
template<class T>
  valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&, const T&);
template<class T> valarray<bool> operator> (const T&, const valarray<T>&);
template<class T>
  valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator<=(const T&, const valarray<T>&);
template<class T>
  valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator>=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator>=(const T&, const valarray<T>&);
template<class T> valarray<T> abs (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const T&);
template<class T> valarray<T> atan2(const T&, const valarray<T>&);
template<class T> valarray<T> cos (const valarray<T>&);
template<class T> valarray<T> cosh (const valarray<T>&);
template<class T> valarray<T> exp (const valarray<T>&);
template<class T> valarray<T> log (const valarray<T>&);
template<class T> valarray<T> log10(const valarray<T>&);
template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> pow(const valarray<T>&, const T&);
template<class T> valarray<T> pow(const T&, const valarray<T>&);
```

§ 26.6.1 932

```
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> 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.
- <sup>2</sup> 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. <sup>280</sup>
- <sup>4</sup> Implementations introducing such replacement types shall provide additional functions and operators as follows:
  - for every function taking a const valarray<T>&, identical functions taking the replacement types shall be added:
  - for every function taking two const valarray<T>& arguments, identical functions taking every combination of const valarray<T>& and replacement types shall be added.
- In particular, an implementation shall allow a valarray<T> to be constructed from such replacement types and shall allow assignments and computed assignments of such types to valarray<T>, slice\_array<T>, gslice\_array<T>, mask\_array<T> and indirect\_array<T> objects.
- These library functions are permitted to throw a bad\_alloc (18.6.2.1) exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

#### 26.6.2 Class template valarray

[template.valarray]

#### 26.6.2.1 Class template valarray overview

[template.valarray.overview]

```
namespace std {
  template<class T> class valarray {
  public:
    typedef T value_type;

    // 26.6.2.2 construct/destroy:
    valarray();
    explicit valarray(size_t);
    valarray(const T&, size_t);
    valarray(const T*, size_t);
    valarray(const valarray&);
    valarray(valarray&&) noexcept;
    valarray(const slice_array<T>&);
    valarray(const gslice_array<T>&);
    valarray(const mask_array<T>&);
    valarray(const mask_array<T>&);
```

280) Clause 18.3.2 recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for valarray expressions.

§ 26.6.2.1

```
valarray(const indirect_array<T>&);
valarray(initializer_list<T>);
~valarray();
// 26.6.2.3 assignment:
valarray<T>& operator=(const valarray<T>&);
valarray<T>& operator=(valarray<T>&&) noexcept;
valarray& operator=(initializer list<T>);
valarray<T>& operator=(const T&);
valarray<T>& operator=(const slice_array<T>&);
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);
// 26.6.2.4 element access:
                 operator[](size_t) const;
const T&
T&
                  operator[](size_t);
// 26.6.2.5 subset operations:
valarray<T>
                operator[](slice) const;
slice_array<T> operator[](slice);
valarray<T>
             operator[](const gslice&) const;
gslice_array<T> operator[](const gslice&);
                  operator[](const valarray<bool>&) const;
valarray<T>
mask_array<T>
                  operator[](const valarray<bool>&);
valarray<T>
                  operator[](const valarray<size_t>&) const;
indirect_array<T> operator[](const valarray<size_t>&);
// 26.6.2.6 unary operators:
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<bool> operator!() const;
// 26.6.2.7 computed assignment:
valarray<T>& operator*= (const T&);
valarray<T>& operator/= (const T&);
valarray<T>& operator%= (const T&);
valarray<T>& operator+= (const T&);
valarray<T>& operator-= (const T&);
valarray<T>& operator^= (const T&);
valarray<T>& operator&= (const T&);
valarray<T>& operator|= (const T&);
valarray<T>& operator<<=(const T&);</pre>
valarray<T>& operator>>=(const T&);
valarray<T>& operator*= (const valarray<T>&);
valarray<T>& operator/= (const valarray<T>&);
valarray<T>& operator%= (const valarray<T>&);
valarray<T>& operator+= (const valarray<T>&);
valarray<T>& operator-= (const valarray<T>&);
valarray<T>& operator^= (const valarray<T>&);
valarray<T>& operator|= (const valarray<T>&);
valarray<T>& operator&= (const valarray<T>&);
valarray<T>& operator<<=(const valarray<T>&);
```

§ 26.6.2.1 934

```
valarray<T>& operator>>=(const valarray<T>&);

// 26.6.2.8 member functions:
void swap(valarray&) noexcept;

size_t size() const;

T sum() const;
T min() const;
T max() const;

valarray<T> shift (int) const;
valarray<T> cshift(int) const;
valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;
void resize(size_t sz, T c = T());
};
}
```

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.<sup>281</sup>

<sup>2</sup> An implementation is permitted to qualify any of the functions declared in <valarray> as inline.

#### 26.6.2.2 valarray constructors

[valarray.cons]

valarray();

1

Effects: Constructs an object of class valarray<T>282 which has zero length. 283

```
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).

```
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.

```
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.

§ 26.6.2.2 935

<sup>281)</sup> The intent is to specify an array template that has the minimum functionality necessary to address aliasing ambiguities and the proliferation of temporaries. Thus, the **valarray** template is neither a matrix class nor a field class. However, it is a very useful building block for designing such classes.

<sup>282)</sup> For convenience, such objects are referred to as "arrays" throughout the remainder of 26.6.

<sup>283)</sup> 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.

<sup>284)</sup> This constructor is the preferred method for converting a C array to a valarray object.

```
valarray(const valarray<T>&);
5
         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.<sup>285</sup>
    valarray(valarray<T>&& v) noexcept;
6
         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.
7
         Complexity: Constant.
    valarray(initializer_list<T> il);
8
         Effects: Same as valarray(il.begin(), il.size()).
    valarray(const slice_array<T>&);
    valarray(const gslice_array<T>&);
    valarray(const mask_array<T>&);
    valarray(const indirect_array<T>&);
9
         These conversion constructors convert one of the four reference templates to a valarray.
    ~valarray();
10
         The destructor is applied to every element of *this; an implementation may return all allocated
    26.6.2.3
             valarray assignment
                                                                                           [valarray.assign]
    valarray<T>& operator=(const valarray<T>& 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
         Postcondition: size() == v.size().
    valarray<T>& operator=(valarray<T>&& v) noexcept;
3
         Effects: *this obtains the value of v. If the length of v is not equal to the length of *this, resizes
         *this to make the two arrays the same length, as if by calling resize(v.size()), before performing
         the assignment.
4
         Complexity: Constant.
    valarray& operator=(initializer_list<T> il);
5
```

Effects: \*this = valarray(il).

Returns: \*this.

6

§ 26.6.2.3 936

<sup>285)</sup> 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.

```
valarray<T>& operator=(const T&);
```

7 The scalar assignment operator causes each element of the \*this array to be assigned the value of the argument.

```
valarray<T>& operator=(const slice_array<T>&);
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);
```

- 8 Requires: The length of the array to which the argument refers equals size().
- These operators allow the results of a generalized subscripting operation to be assigned directly to a valarray.
- If the value of an element in the left-hand side of a valarray assignment operator depends on the value of another element in that left-hand side, the resulting behavior is undefined.

### 26.6.2.4 valarray element access

[valarray.access]

```
const T& operator[](size_t) const;
T& operator[](size_t);
```

1

2

- The subscript operator returns a reference to the corresponding element of the array.
- Thus, the expression (a[i] = q, a[i]) == q evaluates as true for any non-constant valarray<T> a, any T q, and for any size\_t i such that the value of i is less than the length of a.
- 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] != &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. 286
- The reference returned by the subscript operator for an array is guaranteed to be valid until the member function resize(size\_t, T) (26.6.2.8) 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.

# 26.6.2.5 valarray subset operations

[valarray.sub]

The member operator[] is overloaded to provide several ways to select sequences of elements from among those controlled by \*this. Each of these operations returns a subset of the array. The const-qualified versions return this subset as a new valarray object. The non-const versions return a class template object which has reference semantics to the original array, working in conjunction with various overloads of operator= and other assigning operators to allow selective replacement (slicing) of the controlled sequence. In each case the selected element(s) must exist.

```
valarray<T> operator[](slice slicearr) const;
```

Returns: An object of class valarray<T> containing those elements of the controlled sequence designated by slicearr. [Example:

```
const valarray<char> v0("abcdefghijklmnop", 16);
// v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
```

§ 26.6.2.5

<sup>286)</sup> Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from operator new, and other techniques to generate efficient valarrays.

```
— end example]
   slice_array<T> operator[](slice slicearr);
3
        Returns: An object that holds references to elements of the controlled sequence selected by slicearr.
        [Example:
          valarray<char> v0("abcdefghijklmnop", 16);
          valarray<char> v1("ABCDE", 5);
          v0[slice(2, 5, 3)] = v1;
          // v0 == valarray<char>("abAdeBghCjkDmnEp", 16);
         — end example]
   valarray<T> operator[](const gslice& gslicearr) const;
4
        Returns: An object of class valarray<T> containing those elements of the controlled sequence desig-
        nated by gslicearr. [Example:
          const valarray<char> v0("abcdefghijklmnop", 16);
          const size_t lv[] = { 2, 3 };
          const size_t dv[] = { 7, 2 };
          const valarray<size_t> len(lv, 2), str(dv, 2);
          // v0[gslice(3, len, str)] returns
          // valarray<char>("dfhkmo", 6)
         — end example]
   gslice_array<T> operator[](const gslice& gslicearr);
5
        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;
6
        Returns: An object of class valarray<T> containing those elements of the controlled sequence desig-
        nated by boolarr. [Example:
          const valarray<char> v0("abcdefghijklmnop", 16);
          const bool vb[] = { false, false, true, true, false, true };
          // v0[valarray<bool>(vb, 6)] returns
          // valarray<char>("cdf", 3)
         — end example]
   § 26.6.2.5
                                                                                                      938
```

```
mask_array<T> operator[](const valarray<bool>& boolarr);
        Returns: An object that holds references to elements of the controlled sequence selected by boolarr.
        [Example:
          valarray<char> v0("abcdefghijklmnop", 16);
          valarray<char> v1("ABC", 3);
          const bool vb[] = { false, false, true, true, false, true };
          v0[valarray<bool>(vb, 6)] = v1;
          // v0 == valarray<char>("abABeCghijklmnop", 16)
         — end example]
   valarray<T> operator[](const valarray<size_t>& indarr) const;
8
        Returns: An object of class valarray<T> containing those elements of the controlled sequence desig-
        nated by indarr. [Example:
          const valarray<char> v0("abcdefghijklmnop", 16);
          const size_t vi[] = { 7, 5, 2, 3, 8 };
          // v0[valarray<size_t>(vi, 5)] returns
          // valarray<char>("hfcdi", 5)
         — end example
   indirect_array<T> operator[](const valarray<size_t>& indarr);
9
        Returns: An object that holds references to elements of the controlled sequence selected by indarr.
        [Example:
          valarray<char> v0("abcdefghijklmnop", 16);
          valarray<char> v1("ABCDE", 5);
          const size_t vi[] = { 7, 5, 2, 3, 8 };
          v0[valarray<size_t>(vi, 5)] = v1;
          // v0 == valarray<char>("abCDeBgAEjklmnop", 16)
         — end example]
                                                                                        [valarray.unary]
   26.6.2.6 valarray unary operators
   valarray<T> operator+() const;
   valarray<T> operator-() const;
   valarray<T> operator~() const;
   valarray<bool> operator!() const;
        Each of these operators may only be instantiated for a type T to which the indicated operator can be
        applied and for which the indicated operator returns a value which is of type T (bool for operator!)
        or which may be unambiguously implicitly converted to type T (bool for operator!).
        Each of these operators returns an array whose length is equal to the length of the array. Each
```

§ 26.6.2.6 939

corresponding element of the array.

element of the returned array is initialized with the result of applying the indicated operator to the

#### 26.6.2.7 valarray computed assignment

[valarray.cassign]

```
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>&);
valarray<T>& operator<<=(const valarray<T>&);
valarray<T>& operator<>=(const valarray<T>&);
```

Each of these operators may only be instantiated for a type T to which the indicated operator can be applied. Each of these operators performs the indicated operation on each of its elements and the corresponding element of the argument array.

- The array is then returned by reference.
- If the array and the argument array do not have the same length, the behavior is undefined. The appearance of an array on the left-hand side of a computed assignment does **not** invalidate references or pointers.
- If the value of an element in the left-hand side of a valarray computed assignment operator depends on the value of another element in that left hand side, the resulting behavior is undefined.

```
valarray<T>& operator*= (const T&);
valarray<T>& operator/= (const T&);
valarray<T>& operator%= (const T&);
valarray<T>& operator+= (const T&);
valarray<T>& operator-= (const T&);
valarray<T>& operator^= (const T&);
valarray<T>& operator%= (const T&);
valarray<T>& operator%= (const T&);
valarray<T>& operator%= (const T%);
valarray<T>& operator<= (const T%);
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.
- <sup>7</sup> 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.

# 26.6.2.8 valarray member functions

[valarray.members]

```
void swap(valarray& v) noexcept;

Effects: *this obtains the value of v. v obtains the value of *this.

Complexity: Constant.
```

```
size_t size() const;
```

§ 26.6.2.8 940

- 3 Returns: The number of elements in the array.
- 4 Complexity: constant time.

#### T sum() const;

This function may only be instantiated for a type T to which operator+= can be applied. This function returns the sum of all the elements of the array.

If the array has length 0, the behavior is undefined. If the array has length 1, sum() returns the value of element 0. Otherwise, the returned value is calculated by applying operator+= to a copy of an element of the array and all other elements of the array in an unspecified order.

#### T min() const;

This function returns the minimum value contained in \*this. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator<.

### T max() const;

This function returns the maximum value contained in \*this. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator<.

```
valarray<T> shift(int n) const;
```

- This function returns an object of class valarray<T> of length size(), each of whose elements I is (\*this)[I + n] if I + n is non-negative and less than size(), otherwise T(). Thus if element zero is taken as the leftmost element, a positive value of n shifts the elements left n places, with zero fill.
- [Example: If the argument has the value -2, the first two elements of the result will be value-initialized (8.5); the third element of the result will be assigned the value of the first element of the argument; etc. end example]

```
valarray<T> cshift(int n) const;
```

This function returns an object of class valarrayT> 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.

§ 26.6.2.8 941

# 26.6.3 valarray non-member operations 26.6.3.1 valarray binary operators

[valarray.nonmembers] [valarray.binary]

```
template<class T> valarray<T> operator*
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator%
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator-
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator&
    (const valarray<T>&, const valarray<T>&);
template < class T > valarray < T > operator |
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>
    (const valarray<T>&, const valarray<T>&);
```

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 or which can be unambiguously implicitly converted to type T.

Each of these operators returns an array whose length is equal to the lengths of the argument arrays. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

If the argument arrays do not have the same length, the behavior is undefined.

```
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>&);
```

§ 26.6.3.1 942

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

[valarray.comparison]

```
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>&);
```

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 bool or which can be unambiguously implicitly converted to type bool.

- Each of these operators returns a bool array whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.
- If the two array arguments do not have the same length, the behavior is undefined.

```
template<class T> valarray<bool> operator==(const valarray<T>&, const 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>&);
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>&);
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.

§ 26.6.3.2

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

5

1

[valarray.transcend]

```
template<class T> valarray<T> abs (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);
template<class T> valarray<T> atan2
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const T&);
template<class T> valarray<T> atan2(const T&, const valarray<T>&);
template<class T> valarray<T> cos (const valarray<T>&);
template<class T> valarray<T> cosh (const valarray<T>&);
template<class T> valarray<T> exp (const valarray<T>&);
template<class T> valarray<T> log (const valarray<T>&);
template<class T> valarray<T> log10(const valarray<T>&);
template<class T> valarray<T> pow
    (const valarray<T>&, const valarray<T>&);
\label{template} $$ $$ 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>&);
```

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

[valarray.special]

```
template <class T> void swap(valarray<T>& x, valarray<T>& y) noexcept;
Effects: x.swap(y).
```

### 26.6.4 Class slice

[class.slice]

# ${\bf 26.6.4.1} \quad {\bf Class \ slice \ overview}$

[class.slice.overview]

```
namespace std {
  class slice {
   public:
     slice();
     slice(size_t, size_t, size_t);

     size_t start() const;
     size_t size() const;
     size_t stride() const;
};
}
```

§ 26.6.4.1 944

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.<sup>287</sup>

```
26.6.4.2 slice constructors
```

[cons.slice]

```
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]

# 26.6.4.3 slice access functions

2

[slice.access]

```
size_t start() const;
size_t size() const;
size_t stride() const;
Returns: The start, length, or stride specified by a slice object.
```

# 26.6.5 Class template slice\_array

Complexity: constant time.

[template.slice.array]

### 26.6.5.1 Class template slice\_array overview

[template.slice.array.overview]

```
namespace std {
  template <class T> class slice_array {
  public:
    typedef T value_type;
    void operator= (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;
    slice_array(const slice_array&);
    ~slice_array();
    const slice_array& operator=(const slice_array&) const;
  void operator=(const T&) const;
    slice_array() = delete;
                                   // as implied by declaring copy constructor above
  };
}
```

§ 26.6.5.1 945

<sup>287)</sup> 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.

<sup>1</sup> 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]

```
26.6.5.2 slice_array assignment
```

[slice.arr.assign]

```
void operator=(const valarray<T>&) const;
const slice_array& operator=(const slice_array&) const;
```

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which the slice\_array object refers.

### 26.6.5.3 slice\_array computed assignment

[slice.arr.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<=(const valarray<T>&) const;
void operator<=(const valarray<T>&) const;
void operator<>=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the slice\_array object refers.

# 26.6.5.4 slice\_array fill function

[slice.arr.fill]

```
void operator=(const T&) const;
```

1

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the slice\_array object refers.

#### 26.6.6 The gslice class

[class.gslice]

### 26.6.6.1 The gslice class overview

[class.gslice.overview]

This class represents a generalized slice out of an array. A gslice is defined by a starting offset (s), a set of lengths  $(l_i)$ , and a set of strides  $(d_i)$ . The number of lengths shall equal the number of strides.

§ 26.6.6.1 946

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  $l_{ij}-1$ .

<sup>3</sup> [Example: The gslice specification

```
start = 3
length = {2, 4, 3}
stride = {19, 4, 1}
```

yields the sequence of one-dimensional indices

$$k = 3 + (0,1) \times 19 + (0,1,2,3) \times 4 + (0,1,2) \times 1$$

which are ordered as shown in the following table:

$$\begin{array}{rclcrcr} (i_0, & i_1, & i_2, & k) & = \\ & (0, & 0, & 0, & 3), \\ & (0, & 0, & 1, & 4), \\ & (0, & 0, & 2, & 5), \\ & (0, & 1, & 0, & 7), \\ & (0, & 1, & 1, & 8), \\ & (0, & 1, & 2, & 9), \\ & (0, & 2, & 0, & 11), \\ & (0, & 2, & 1, & 12), \\ & (0, & 2, & 2, & 13), \\ & (0, & 3, & 0, & 15), \\ & (0, & 3, & 1, & 16), \\ & (0, & 3, & 2, & 17), \\ & (1, & 0, & 0, & 22), \\ & (1, & 0, & 1, & 23), \\ & \cdots & & & \\ & (1, & 3, & 2, & 36) \\ \end{array}$$

That is, the highest-ordered index turns fastest. — end example

- <sup>4</sup> It is possible to have degenerate generalized slices in which an address is repeated.
- Example: If the stride parameters in the previous example are changed to  $\{1, 1, 1\}$ , the first few elements of the resulting sequence of indices will be

— end example]

<sup>6</sup> If a degenerate slice is used as the argument to the non-const version of operator[] (const gslice&), the resulting behavior is undefined.

§ 26.6.6.1 947

```
26.6.6.2 gslice constructors
                                                                                          [gslice.cons]
gslice();
gslice(size_t start, const valarray<size_t>& lengths,
         const valarray<size_t>& strides);
gslice(const gslice&);
     The default constructor is equivalent to gslice(0, valarray<size_t>(), valarray<size_t>()).
     The constructor with arguments builds a gslice based on a specification of start, lengths, and strides,
     as explained in the previous section.
                                                                                        [gslice.access]
26.6.6.3 gslice access functions
size t
                 start() const;
valarray<size_t> size() const;
valarray<size_t> stride() const;
     Returns: The representation of the start, lengths, or strides specified for the gslice.
      Complexity: start() is constant time. size() and stride() are linear in the number of strides.
26.6.7 Class template gslice_array
                                                                            [template.gslice.array]
                                                                    [template.gslice.array.overview]
26.6.7.1
          Class template gslice_array overview
  namespace std {
    template <class T> class gslice_array {
      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.
```

2

2

3

§ 26.6.7.1 948

```
26.6.7.2 gslice_array assignment
```

[gslice.array.assign]

```
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.

### 26.6.7.3 gslice\_array

1

[gslice.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<=(const valarray<T>&) const;
void operator<=(const valarray<T>&) const;
void operator>=(const valarray<T>&) const;
void operator>=(const valarray<T>&) const;
```

These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the gslice\_array object refers.

### 26.6.7.4 gslice\_array fill function

[gslice.array.fill]

void operator=(const T&) const;

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the gslice\_array object refers.

# 26.6.8 Class template mask\_array

[template.mask.array]

# 26.6.8.1 Class template mask\_array overview

[template.mask.array.overview]

```
namespace std {
  template <class T> class mask_array {
  public:
    typedef T value_type;
    void operator= (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator = (const valarray < T > &) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;
    mask_array(const mask_array&);
   ~mask_array();
    const mask_array& operator=(const mask_array&) const;
    void operator=(const T&) const;
```

§ 26.6.8.1 949

```
mask_array() = delete;  // as implied by declaring copy constructor above
};
}
```

This template is a helper template used by the mask subscript operator:

```
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.)

# 26.6.8.2 mask\_array assignment

2

[mask.array.assign]

```
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.

#### 26.6.8.3 mask\_array computed assignment

[mask.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
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.

# 26.6.8.4 mask\_array fill function

[mask.array.fill]

```
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

[template.indirect.array]

### 26.6.9.1 Class template indirect\_array overview

[template.indirect.array.overview]

```
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;
```

§ 26.6.9.1 950

1 This template is a helper template used by the indirect subscript operator

indirect\_array<T> valarray<T>::operator[](const valarray<size\_t>&).

It has reference semantics to a subset of an array specified by an indirect\_array. Thus the expression a[indirect] = b; has the effect of assigning the elements of b to the elements in a whose indices appear in indirect.

### 26.6.9.2 indirect\_array assignment

[indirect.array.assign]

```
void operator=(const valarray<T>&) const;
const indirect_array& operator=(const indirect_array&) const;
```

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which it refers.

- If the indirect\_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.
- 3 [Example:

2

1

```
int addr[] = {2, 3, 1, 4, 4};
valarray<size_t> indirect(addr, 5);
valarray<double> a(0., 10), b(1., 5);
a[indirect] = b;
```

results in undefined behavior since element 4 is specified twice in the indirection. — end example

# 26.6.9.3 indirect\_array computed assignment

[indirect.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<=(const valarray<T>&) const;
void operator<>=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the indirect\_array object refers.

If the indirect\_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.

§ 26.6.9.3 951

#### 26.6.9.4 indirect\_array fill function

[indirect.array.fill]

```
void operator=(const T&) const;
```

1

2

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the indirect\_array object refers.

#### 26.6.10 valarray range access

[valarray.range]

In the begin and end function templates that follow, unspecified 1 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&. unspecified 2 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

[numeric.ops]

## 26.7.1 Header <numeric> synopsis

[numeric.ops.overview]

```
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>

§ 26.7.1 952

```
OutputIterator adjacent_difference(InputIterator first,
                                          InputIterator last,
                                          OutputIterator result);
    template <class InputIterator, class OutputIterator,
               class BinaryOperation>
      OutputIterator adjacent_difference(InputIterator first,
                                          InputIterator last,
                                          OutputIterator result,
                                          BinaryOperation binary_op);
    template <class ForwardIterator, class T>
      void iota(ForwardIterator first, ForwardIterator last, T value);
The requirements on the types of algorithms' arguments that are described in the introduction to Clause 25
also apply to the following algorithms.
26.7.2 Accumulate
                                                                                        [accumulate]
template <class InputIterator, class T>
  T accumulate(InputIterator first, InputIterator last, T init);
template <class InputIterator, class T, class BinaryOperation>
  T accumulate(InputIterator first, InputIterator last, T init,
                BinaryOperation binary_op);
      Effects: Computes its result by initializing the accumulator acc with the initial value init and then
      modifies it with acc = acc + *i or acc = binary_op(acc, *i) for every iterator i in the range
      [first,last) in order.<sup>288</sup>
      Requires: T shall meet the requirements of CopyConstructible (Table 21) and CopyAssignable (Ta-
      ble 23) types. In the range [first,last], binary_op shall neither modify elements nor invalidate
      iterators or subranges.<sup>289</sup>
26.7.3
         Inner product
                                                                                     [inner.product]
template <class InputIterator1, class InputIterator2, class T>
  T inner_product(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, T init);
template <class InputIterator1, class InputIterator2, class T,
    class BinaryOperation1, class BinaryOperation2>
  T inner_product(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, T init,
                   BinaryOperation1 binary_op1,
                   BinaryOperation2 binary_op2);
      Effects: Computes its result by initializing the accumulator acc with the initial value init and
      then modifying it with acc = acc + (*i1) * (*i2) or acc = binary_op1(acc, binary_op2(*i1,
      *i2)) for every iterator i1 in the range [first1,last1) and iterator i2 in the range [first2,first2
      + (last1 - first1)) in order.
```

288) 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.

op1 and binary\_op2 shall neither modify elements nor invalidate iterators or subranges.<sup>290</sup>

Requires: T shall meet the requirements of CopyConstructible (Table 21) and CopyAssignable (Table 23) types. In the ranges [first1,last1] and [first2,first2 + (last1 - first1)] binary\_-

1

2

1

2

§ 26.7.3 953

<sup>289)</sup> The use of fully closed ranges is intentional

<sup>290)</sup> The use of fully closed ranges is intentional

26.7.4 Partial sum [partial.sum]

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)).

2 Returns: result + (last - first).

1

1

- 3 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.<sup>291</sup>
- 5 Remarks: result may be equal to first.

# 26.7.5 Adjacent difference

[adjacent.difference]

```
template <class InputIterator, class OutputIterator>
  OutputIterator adjacent_difference(
    InputIterator first, InputIterator last,
    OutputIterator result);
template <class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator adjacent_difference(
    InputIterator first, InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);
    Effects: For a non-empty range, the function creates an accumulator acc who
```

Effects: For a non-empty range, the function creates an accumulator acc whose type is InputIterator's value type, initializes it with \*first, and assigns the result to \*result. For every iterator i in [first + 1,last) in order, creates an object val whose type is InputIterator's value type, initializes it with \*i, computes val - acc or binary\_op(val, acc), assigns the result to \*(result + (i - first)), and move assigns from val to acc.

Requires: InputIterator's value type shall be MoveAssignable (Table 22) and shall be constructible from the type of \*first. acc shall be writable to the result output iterator. The result of the expression val - acc or binary\_op(val, acc) shall be writable to the result output iterator. In the ranges [first,last] and [result,result + (last - first)], binary\_op shall neither modify elements nor invalidate iterators or subranges.<sup>292</sup>

- 2 Remarks: result may be equal to first.
- 3 Returns: result + (last first).
- 4 Complexity: Exactly (last first) 1 applications of the binary operation.

§ 26.7.5

<sup>291)</sup> The use of fully closed ranges is intentional.

<sup>292)</sup> The use of fully closed ranges is intentional.

26.7.6 Iota [numeric.iota]

template <class ForwardIterator, class T>

void iota(ForwardIterator first, ForwardIterator last, T value);

Requires: T shall be convertible to ForwardIterator's value type. The expression ++val, where val has type T, shall be well formed.

- 2 Effects: For each element referred to by the iterator i in the range [first,last), assigns \*i = value and increments value as if by ++value.
- 3 Complexity: Exactly last first increments and assignments.

26.8 C library [c.math]

- The header <ctgmath> simply includes the headers <ccomplex> and <cmath>.
- <sup>2</sup> [Note: The overloads provided in C by magic macros are already provided in <ccomplex> and <cmath> by "sufficient" additional overloads. end note]
- 3 Tables 119 and 120 describe headers <cmath> and <cstdlib>, respectively.

Type	$\mathbf{Name}(\mathbf{s})$					
Macros:						
FP_FAST_FMA	FP_ILOGBNAN	FP_SUBNORMAL	HUGE_VALL	MATH_ERRNO		
FP_FAST_FMAF	FP_INFINITE	FP_ZERO	INFINITY	MATH_ERREXCEPT		
FP_FAST_FMAL	FP_NAN	HUGE_VAL	NAN	math_errhandling		
FP_ILOGBO	FP_NORMAL	HUGE_VALF				
Types:	double_t	float_t				
Math Functions:						
abs	cosh	fmod	logb	remquo		
acos	erf	frexp	lrint	rint		
acosh	erfc	hypot	lround	round		
asin	exp2	ilogb	modf	scalbln		
asinh	exp	ldexp	nan	scalbn		
atan	expm1	lgamma	nanf	sin		
atan2	fabs	llrint	nanl	sinh		
atanh	fdim	llround	${\tt nearbyint}$	sqrt		
cbrt	floor	log	nextafter	tan		
ceil	fma	log10	nexttoward	tanh		
copysign	fmax	log1p	pow	tgamma		
cos	fmin	log2	remainder	trunc		
,	Classification/comparison Functions:					
fpclassify	isgreaterequal	islessequal	isnan	isunordered		
isfinite	isinf	islessgreater	isnormal	signbit		
isgreater	isless					

Table 119 — Header <cmath> synopsis

<sup>4</sup> 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 (17.6.5.9). [Note: The random number generation (26.5) facilities in this standard are often preferable to rand. — end note]

Table 120 —	Header	<cstdlib></cstdlib>	synopsis

Type	Name(s)		
Macro:	RAND_MAX		
Types:			
div_t	ldiv_t	lldiv_t	
Functions:			
abs	ldiv	rand	
div	llabs	srand	
labs	lldiv		

- <sup>6</sup> In addition to the int versions of certain math functions in <cstdlib>, C++ adds long and long overloaded versions of these functions, with the same semantics.
- <sup>7</sup> The added signatures are:

- <sup>8</sup> 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.
- <sup>9</sup> The added signatures are:

```
float abs(float);
float acos(float);
float acosh(float);
float asin(float);
float asinh(float);
float atan(float);
float atan2(float, float);
float atanh(float);
float cbrt(float);
float ceil(float);
float copysign(float, float);
float cos(float);
float cosh(float);
float erf(float);
float erfc(float);
float exp(float);
float exp2(float);
float expm1(float);
float fabs(float);
float fdim(float, float);
float floor(float);
float fma(float, float, float);
float fmax(float, float);
float fmin(float, float);
float fmod(float, float);
float frexp(float, int*);
float hypot(float, float);
int ilogb(float);
float ldexp(float, int);
```

```
float lgamma(float);
long long llrint(float);
long long llround(float);
float log(float);
float log10(float);
float log1p(float);
float log2(float);
float logb(float);
long lrint(float);
long lround(float);
float modf(float, float*);
float nearbyint(float);
float nextafter(float, float);
float nexttoward(float, long double);
float pow(float, float);
float remainder(float, float);
float remquo(float, float, int *);
float rint(float);
float round(float);
float scalbln(float, long);
float scalbn(float, int);
float sin(float);
float sinh(float);
float sqrt(float);
float tan(float);
float tanh(float);
float tgamma(float);
float trunc(float);
double abs(double);
                               // fabs()
long double abs(long double);
long double acos(long double);
long double acosh(long double);
long double asin(long double);
long double asinh(long double);
long double atan(long double);
long double atan2(long double, long double);
long double atanh(long double);
long double cbrt(long double);
long double ceil(long double);
long double copysign(long double, long double);
long double cos(long double);
long double cosh(long double);
long double erf(long double);
long double erfc(long double);
long double exp(long double);
long double exp2(long double);
long double expm1(long double);
long double fabs(long double);
long double fdim(long double, long double);
long double floor(long double);
long double fma(long double, long double, long double);
long double fmax(long double, long double);
long double fmin(long double, long double);
```

```
long double fmod(long double, long double);
long double frexp(long double, int*);
long double hypot(long double, long double);
int ilogb(long double);
long double ldexp(long double, int);
long double lgamma(long double);
long long llrint(long double);
long long llround(long double);
long double log(long double);
long double log10(long double);
long double log1p(long double);
long double log2(long double);
long double logb(long double);
long lrint(long double);
long lround(long double);
long double modf(long double, long double*);
long double nearbyint(long double);
long double nextafter(long double, long double);
long double nexttoward(long double, long double);
long double pow(long double, long double);
long double remainder(long double, long double);
long double remquo(long double, long double, int *);
long double rint(long double);
long double round(long double);
long double scalbln(long double, long);
long double scalbn(long double, int);
long double sin(long double);
long double sinh(long double);
long double sqrt(long double);
long double tan(long double);
long double tanh(long double);
long double tgamma(long double);
long double trunc(long double);
```

The classification/comparison functions behave the same as the C macros with the corresponding names defined in 7.12.3, Classification macros, and 7.12.14, Comparison macros in the C Standard. Each function is overloaded for the three floating-point types, as follows:

```
int fpclassify(float x);
bool isfinite(float x);
bool isinf(float x);
bool isnan(float x);
bool isnormal(float x);
bool signbit(float x);
bool isgreater(float x, float y);
bool isgreaterequal(float x, float y);
bool isless(float x, float y);
bool islessequal(float x, float y);
bool islessgreater(float x, float y);
bool isunordered(float x, float y);
int fpclassify(double x);
bool isfinite(double x);
bool isinf(double x);
bool isnan(double x);
```

```
bool isnormal(double x);
bool signbit(double x);
bool isgreater(double x, double y);
bool isgreaterequal(double x, double y);
bool isless(double x, double y);
bool islessequal(double x, double y);
bool islessgreater(double x, double y);
bool isunordered(double x, double y);
int fpclassify(long double x);
bool isfinite(long double x);
bool isinf(long double x);
bool isnan(long double x);
bool isnormal(long double x);
bool signbit(long double x);
bool isgreater(long double x, long double y);
bool isgreaterequal(long double x, long double y);
bool isless(long double x, long double y);
bool islessequal(long double x, long double y);
bool islessgreater(long double x, long double y);
bool isunordered(long double x, long double y);
```

- 11 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.

SEE ALSO: ISO C 7.5, 7.10.2, 7.10.6.

§ 26.8 959

## 27 Input/output library

# [input.output]

## 27.1 General

[input.output.general]

- This Clause describes components that C++ programs may use to perform input/output operations.
- <sup>2</sup> The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 121.

	Subclause	Header(s)
27.2	Requirements	
27.3	Forward declarations	<iosfwd></iosfwd>
27.4	Standard iostream objects	<iostream></iostream>
27.5	Iostreams base classes	<ios></ios>
27.6	Stream buffers	<streambuf></streambuf>
27.7	Formatting and manipulators	<istream></istream>
		<pre><ostream></ostream></pre>
		<iomanip></iomanip>
27.8	String streams	<sstream></sstream>
27.9	File streams	<fstream></fstream>
		<cstdio></cstdio>
		<cinttypes></cinttypes>

Table 121 — Input/output library summary

Figure 7 illustrates relationships among various types described in this clause. A line from  $\bf A$  to  $\bf B$  indicates that  $\bf A$  is an alias (e.g. a typedef) for  $\bf B$  or that  $\bf A$  is defined in terms of  $\bf B$ .

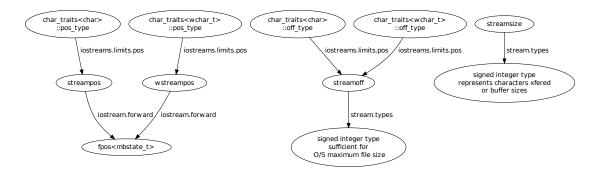


Figure 7 — Stream position, offset, and size types [non-normative]

## 27.2 Iostreams requirements

## [iostreams.requirements] [iostream.limits.imbue]

## 27.2.1 Imbue limitations

No function described in Clause 27 except for ios\_base::imbue and basic\_filebuf::pubimbue causes any

§ 27.2.1 960

instance of basic\_ios::imbue or basic\_streambuf::imbue to be called. If any user function called from a function declared in Clause 27 or as an overriding virtual function of any class declared in Clause 27 calls imbue, the behavior is undefined.

## 27.2.2 Positioning type limitations

## [iostreams.limits.pos]

- The classes of Clause 27 with template arguments charT and traits behave as described if traits::pos\_type and traits::off\_type are streampos and streamoff respectively. Except as noted explicitly below, their behavior when traits::pos\_type and traits::off\_type are other types is implementation-defined.
- <sup>2</sup> In the classes of Clause 27, a template formal parameter with name charT represents a member of the set of types containing char, wchar\_t, and any other implementation-defined character types that satisfy the requirements for a character on which any of the iostream components can be instantiated.

#### 27.2.3 Thread safety

## [iostreams.threadsafety]

- Concurrent access to a stream object (27.8, 27.9), stream buffer object (27.6), or C Library stream (27.9.2) by multiple threads may result in a data race (1.10) unless otherwise specified (27.4). [Note: Data races result in undefined behavior (1.10). end note]
- If one thread makes a library call a that writes a value to a stream and, as a result, another thread reads this value from the stream through a library call b such that this does not result in a data race, then a's write synchronizes with b's read.

#### 27.3 Forward declarations

[iostream.forward]

## Header <iosfwd> synopsis

```
namespace std {
  template<class charT> class char_traits;
  template<> class char_traits<char>;
  template<> class char_traits<char16_t>;
 template<> class char_traits<char32_t>;
  template<> class char_traits<wchar_t>;
 template<class T> class allocator;
  template <class charT, class traits = char_traits<charT> >
    class basic ios:
  template <class charT, class traits = char_traits<charT> >
    class basic_streambuf;
  template <class charT, class traits = char_traits<charT> >
    class basic_istream;
  template <class charT, class traits = char_traits<charT> >
    class basic_ostream;
  template <class charT, class traits = char_traits<charT> >
    class basic_iostream;
  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
    class basic_stringbuf;
  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
    class basic_istringstream;
  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
    class basic_ostringstream;
  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
    class basic_stringstream;
```

§ 27.3 961

```
template <class charT, class traits = char_traits<charT> >
  class basic_filebuf;
template <class charT, class traits = char_traits<charT> >
  class basic_ifstream;
template <class charT, class traits = char_traits<charT> >
  class basic_ofstream;
template <class charT, class traits = char_traits<charT> >
  class basic_fstream;
template <class charT, class traits = char_traits<charT> >
  class istreambuf_iterator;
template <class charT, class traits = char_traits<charT> >
  class ostreambuf_iterator;
typedef basic_ios<char>
                              ios:
typedef basic_ios<wchar_t>
                              wios;
typedef basic_streambuf<char> streambuf;
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
typedef basic_iostream<char> iostream;
typedef basic_stringbuf<char>
                                  stringbuf;
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;
typedef basic_filebuf<char> filebuf;
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;
typedef basic_streambuf<wchar_t> wstreambuf;
typedef basic_istream<wchar_t> wistream;
typedef basic_ostream<wchar_t> wostream;
typedef basic_iostream<wchar_t> wiostream;
typedef basic_stringbuf<wchar_t>
                                     wstringbuf;
typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t> wstringstream;
typedef basic_filebuf<wchar_t> wfilebuf;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_ofstream<wchar_t> wofstream;
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.<sup>293</sup>

§ 27.3

<sup>293)</sup> It is the implementation's responsibility to implement headers so that including <iosfwd> and other headers does not

[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 class templates basic\_stringbuf and basic\_filebuf.
- <sup>4</sup> The class template specialization basic\_istream<charT,traits> serves as a base class for class templates basic\_istringstream and basic\_ifstream.
- <sup>5</sup> The class template specialization basic\_ostream<charT, traits> serves as a base class for class templates basic ostringstream and basic ofstream.
- The class template specialization basic\_iostream<charT, traits> serves as a base class for class templates basic\_stringstream and basic\_fstream.
- 7 Other typedefs define instances of class templates specialized for char or wchar\_t types.
- 8 Specializations of the class template fpos are used for specifying file position information.
- 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

#### 27.4 Standard iostream objects

 $[{\bf iostream.objects}]$ 

## 27.4.1 Overview

[iostream.objects.overview]

## Header <iostream> synopsis

```
#include <ios>
#include <streambuf>
#include <istream>
#include <ostream>

namespace std {
   extern istream cin;
   extern ostream cout;
   extern ostream cerr;
   extern ostream wcin;
   extern wistream wcin;
   extern wostream wcout;
   extern wostream wcerr;
   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.

violate the rules about multiple occurrences of default arguments.

§ 27.4.1 963

The objects are constructed and the associations are established at some time prior to or during the first time an object of class ios\_base::Init is constructed, and in any case before the body of main begins execution.<sup>294</sup> The objects are not destroyed during program execution.<sup>295</sup> 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.

- <sup>3</sup> 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.
- <sup>4</sup> Concurrent access to a synchronized (27.5.3.4) standard iostream object's formatted and unformatted input (27.7.2.1) and output (27.7.3.1) functions or a standard C stream by multiple threads shall not result in a data race (1.10). [Note: Users must still synchronize concurrent use of these objects and streams by multiple threads if they wish to avoid interleaved characters. end note]

## 27.4.2 Narrow stream objects

[narrow.stream.objects]

istream cin;

1

- The object cin controls input from a stream buffer associated with the object stdin, declared in <cstdio>.
- After the object cin is initialized, cin.tie() returns &cout. Its state is otherwise the same as required for basic\_ios<char>::init (27.5.5.2).

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 (27.5.5.2).

ostream clog;

The object clog controls output to a stream buffer associated with the object stderr, declared in <cstdio> (27.9.2).

## 27.4.3 Wide stream objects

[wide.stream.objects]

wistream wcin;

- The object wcin controls input from a stream buffer associated with the object stdin, declared in <cstdio>.
- After the object wcin is initialized, wcin.tie() returns &wcout. Its state is otherwise the same as required for basic\_ios<wchar\_t>::init (27.5.5.2).

§ 27.4.3 964

<sup>294)</sup> If it is possible for them to do so, implementations are encouraged to initialize the objects earlier than required.

<sup>295)</sup> Constructors and destructors for static objects can access these objects to read input from stdin or write output to stdout or stderr.

```
wostream wcout;
```

The object wcout controls output to a stream buffer associated with the object stdout, declared in <cstdio> (27.9.2).

#### wostream wcerr;

- The object wcerr controls output to a stream buffer associated with the object stderr, declared in <cstdio> (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 (27.5.5.2).

wostream wclog;

The object wclog controls output to a stream buffer associated with the object stderr, declared in <cstdio> (27.9.2).

## 27.5 Iostreams base classes

[iostreams.base]

#### 27.5.1 Overview

[iostreams.base.overview]

Header <ios> synopsis

```
#include <iosfwd>
namespace std {
  typedef implementation-defined streamoff;
  typedef implementation-defined streamsize;
  template <class stateT> class fpos;
  class ios_base;
  template <class charT, class traits = char_traits<charT> >
    class basic_ios;
  // 27.5.6, manipulators:
  ios_base& boolalpha (ios_base& str);
 ios_base& noboolalpha(ios_base& str);
  ios_base& showbase
                       (ios_base& str);
  ios_base& noshowbase (ios_base& str);
  ios_base& showpoint (ios_base& str);
 ios_base& noshowpoint(ios_base& str);
  ios_base& showpos
                       (ios_base& str);
  ios_base& noshowpos (ios_base& str);
  ios_base& skipws
                       (ios_base& str);
  ios_base& noskipws
                       (ios_base& str);
  ios_base& uppercase (ios_base& str);
 ios_base& nouppercase(ios_base& str);
 ios_base& unitbuf
                       (ios_base& str);
```

§ 27.5.1 965

```
ios_base& nounitbuf (ios_base& str);
    // 27.5.6.2 adjustfield:
    ios_base& internal
                          (ios_base& str);
    ios_base& left
                          (ios_base& str);
    ios_base& right
                          (ios_base& str);
    // 27.5.6.3 basefield:
    ios_base& dec
                          (ios_base& str);
    ios_base& hex
                          (ios_base& str);
    ios_base& oct
                          (ios_base& str);
    // 27.5.6.4 floatfield:
    ios_base& fixed
                          (ios_base& str);
    ios_base& scientific (ios_base& str);
    ios_base& hexfloat
                          (ios_base& str);
    ios_base& defaultfloat(ios_base& str);
    // 27.5.6.5 error reporting:
    enum class io_errc {
      stream = 1
    };
    template <> struct is_error_code_enum<io_errc> : public true_type { };
    error_code make_error_code(io_errc e);
    error_condition make_error_condition(io_errc e);
    const error_category& iostream_category();
27.5.2
          Types
                                                                                        [stream.types]
typedef implementation-defined streamoff;
     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.<sup>296</sup>
```

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.<sup>297</sup>

#### 27.5.3Class ios base

1

[ios.base]

```
namespace std {
  class ios_base {
 public:
    class failure;
    // 27.5.3.1.2 fmtflags
    typedef T1 fmtflags;
    static constexpr fmtflags boolalpha = unspecified ;
```

§ 27.5.3 966

<sup>296)</sup> Typically long long.

<sup>297)</sup> streamsize is used in most places where ISO C would use size\_t. Most of the uses of streamsize could use size\_t, except for the strstreambuf constructors, which require negative values. It should probably be the signed type corresponding to size\_t (which is what Posix.2 calls ssize\_t).

```
static constexpr fmtflags dec = unspecified ;
static constexpr fmtflags fixed = unspecified ;
static constexpr fmtflags hex = unspecified ;
static constexpr fmtflags internal = unspecified ;
static constexpr fmtflags left = unspecified ;
static constexpr fmtflags oct = unspecified ;
static constexpr fmtflags right = unspecified ;
static constexpr fmtflags scientific = unspecified ;
static constexpr fmtflags showbase = unspecified ;
static constexpr fmtflags showpoint = unspecified ;
static constexpr fmtflags showpos = unspecified ;
static constexpr fmtflags skipws = unspecified ;
static constexpr fmtflags unitbuf = unspecified ;
static constexpr fmtflags uppercase = unspecified ;
static constexpr fmtflags adjustfield = see below;
static constexpr fmtflags basefield = see below;
static constexpr fmtflags floatfield = see below;
// 27.5.3.1.3 iostate
typedef T2 iostate;
static constexpr iostate badbit = unspecified ;
static constexpr iostate eofbit = unspecified ;
static constexpr iostate failbit = unspecified ;
static constexpr iostate goodbit = see below;
// 27.5.3.1.4 openmode
typedef T3 openmode;
static constexpr openmode app = unspecified ;
static constexpr openmode ate = unspecified ;
static constexpr openmode binary = unspecified ;
static constexpr openmode in = unspecified ;
static constexpr openmode out = unspecified ;
static constexpr openmode trunc = unspecified ;
// 27.5.3.1.5 seekdir
typedef T4 seekdir;
static constexpr fmtflags beg = unspecified ;
static constexpr fmtflags cur = unspecified ;
static constexpr fmtflags end = unspecified ;
class Init;
// 27.5.3.2 fmtflags state:
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl, fmtflags mask);
void unsetf(fmtflags mask);
streamsize precision() const;
streamsize precision(streamsize prec);
streamsize width() const;
streamsize width(streamsize wide);
// 27.5.3.3 locales:
```

§ 27.5.3 967

```
locale imbue(const locale& loc);
      locale getloc() const;
      // 27.5.3.5 storage:
      static int xalloc();
      long& iword(int index);
      void*& pword(int index);
      // destructor
      virtual ~ios_base();
      // 27.5.3.6 callbacks;
      enum event { erase_event, imbue_event, copyfmt_event };
      typedef void (*event_callback)(event, ios_base&, int index);
      void register_callback(event_callback fn, int index);
      ios_base(const ios_base&) = delete;
      ios_base& operator=(const ios_base&) = delete;
      static bool sync_with_stdio(bool sync = true);
    protected:
      ios_base();
    private:
      static int index; // exposition only
                         // exposition only
      long* iarray;
      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.
- <sup>2</sup> 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

§ 27.5.3 968

```
27.5.3.1 Types
                                                                                            [ios.types]
27.5.3.1.1
                                                                                          [ios::failure]
             Class ios base::failure
  namespace std {
    class ios_base::failure : public system_error {
    public:
      explicit failure(const string& msg, const error_code& ec = io_errc::stream);
      explicit failure(const char* msg, const error_code& ec = io_errc::stream);
    };
  }
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.
27.5.3.1.2 Type ios_base::fmtflags
                                                                                        [ios::fmtflags]
typedef T1 fmtflags;
     The type fmtflags is a bitmask type (17.5.2.1.3). Setting its elements has the effects indicated in
     Table 122.
     Type fmtflags also defines the constants indicated in Table 123.
                                                                                          [ios::iostate]
27.5.3.1.3 Type ios_base::iostate
typedef T2 iostate;
     The type iostate is a bitmask type (17.5.2.1.3) that contains the elements indicated in Table 124.
     Type iostate also defines the constant:
       — goodbit, the value zero.
27.5.3.1.4 Type ios_base::openmode
                                                                                      [ios::openmode]
typedef T3 openmode;
     The type openmode is a bitmask type (17.5.2.1.3). It contains the elements indicated in Table 125.
27.5.3.1.5 Type ios_base::seekdir
                                                                                         [ios::seekdir]
typedef T4 seekdir;
     The type seekdir is an enumerated type (17.5.2.1.2) that contains the elements indicated in Table 126.
```

3

1

1

2

1

§ 27.5.3.1.5

Table 122 — fmtflags effects

Element	Effect(s) if set
boolalpha	insert and extract bool type in alphabetic format
dec	converts integer input or generates integer output in decimal base
fixed	generate floating-point output in fixed-point notation
hex	converts integer input or generates integer output in hexadecimal base
internal	adds fill characters at a designated internal point in certain generated out-
	put, or identical to right if no such point is designated
left	adds fill characters on the right (final positions) of certain generated output
oct	converts integer input or generates integer output in octal base
right	adds fill characters on the left (initial positions) of certain generated output
scientific	generates floating-point output in scientific notation
showbase	generates a prefix indicating the numeric base of generated integer output
showpoint	generates a decimal-point character unconditionally in generated floating-
	point output
showpos	generates a + sign in non-negative generated numeric output
skipws	skips leading whitespace before certain input operations
unitbuf	flushes output after each output operation
uppercase	replaces certain lowercase letters with their uppercase equivalents in gen-
	erated output

Table 123 — fmtflags constants

Constant	Allowable values
adjustfield	left   right   internal
basefield	dec   oct   hex
floatfield	scientific   fixed

Table 124 — iostate effects

Element	Effect(s) if set
badbit	indicates a loss of integrity in an input or output sequence (such as an
	irrecoverable read error from a file);
eofbit	indicates that an input operation reached the end of an input sequence;
failbit	indicates that an input operation failed to read the expected characters, or
	that an output operation failed to generate the desired characters.

Table 125 — openmode effects

Element	Effect(s) if set
app	seek to end before each write
ate	open and seek to end immediately after opening
binary	perform input and output in binary mode (as opposed to text mode)
in	open for input
out	open for output
trunc	truncate an existing stream when opening

§ 27.5.3.1.5

Table 126 — seekdir effects

Element	Meaning
beg	request a seek (for subsequent input or output) relative to the beginning of the stream
cur	request a seek relative to the current position within the sequence
end	request a seek relative to the current end of the sequence

```
27.5.3.1.6 Class ios_base::Init
```

[ios::Init]

```
namespace std {
  class ios_base::Init {
  public:
     Init();
     ~Init();
  private:
     static int init_cnt; // exposition only
  };
}
```

- The class Init describes an object whose construction ensures the construction of the eight objects declared in <iostream> (27.4) that associate file stream buffers with the standard C streams provided for by the functions declared in <cstdio> (27.9.2).
- <sup>2</sup> For the sake of exposition, the maintained data is presented here as:
  - static int init\_cnt, counts the number of constructor and destructor calls for class Init, initialized to zero.

Init();

Effects: Constructs an object of class Init. Constructs and initializes the objects cin, cout, cerr, clog, wcin, wcout, wcerr, and wclog if they have not already been constructed and initialized.

~Init();

Effects: Destroys an object of class Init. If there are no other instances of the class still in existence, calls cout.flush(), cerr.flush(), clog.flush(), wcout.flush(), wcerr.flush(), wclog.flush().

## 27.5.3.2 ios\_base state functions

[fmtflags.state]

fmtflags flags() const;

Returns: The format control information for both input and output.

fmtflags flags(fmtflags fmtfl);

- Postcondition: fmtfl == flags().
- 3 Returns: The previous value of flags().

fmtflags setf(fmtflags fmtfl);

§ 27.5.3.2 971

```
4
         Effects: Sets fmtfl in flags().
5
         Returns: The previous value of flags().
    fmtflags setf(fmtflags fmtfl, fmtflags mask);
6
         Effects: Clears mask in flags(), sets fmtfl & mask in flags().
7
         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);
10
         Postcondition: prec == precision().
11
         Returns: The previous value of precision().
    streamsize width() const;
12
         Returns: The minimum field width (number of characters) to generate on certain output conversions.
    streamsize width(streamsize wide);
13
         Postcondition: wide == width().
14
         Returns: The previous value of width().
    27.5.3.3
              ios_base functions
                                                                                          [ios.base.locales]
    locale imbue(const locale& loc);
         Effects: Calls each registered callback pair (fn,index) (27.5.3.6) as (*fn) (imbue_event,*this,index)
         at such a time that a call to ios_base::getloc() from within fn returns the new locale value loc.
2
         Returns: The previous value of getloc().
3
         Postcondition: loc == getloc().
    locale getloc() const;
4
         Returns: If no locale has been imbued, a copy of the global C++ locale, locale(), in effect at the time
         of construction. Otherwise, returns the imbued locale, to be used to perform locale-dependent input
         and output operations.
```

§ 27.5.3.3 972

#### 27.5.3.4 ios\_base static members

1

[ios.members.static]

bool sync\_with\_stdio(bool sync = true);

Returns: true if the previous state of the standard iostream objects (27.4) was synchronized and otherwise returns false. The first time it is called, the function returns true.

- <sup>2</sup> Effects: If any input or output operation has occurred using the standard streams prior to the call, the effect is implementation-defined. Otherwise, called with a false argument, it allows the standard streams to operate independently of the standard C streams.
- When a standard iostream object str is synchronized with a standard stdio stream f, the effect of inserting a character c by

```
fputc(f, c);
is the same as the effect of
    str.rdbuf()->sputc(c);
for any sequences of characters; the effect of extracting a character c by
    c = fgetc(f);
is the same as the effect of
    c = str.rdbuf()->sbumpc(c);
for any sequences of characters; and the effect of pushing back a character c by
    ungetc(c, f);
is the same as the effect of
    str.rdbuf()->sputbackc(c);
for any sequence of characters.<sup>298</sup>
```

#### 27.5.3.5 ios base storage functions

[ios.base.storage]

```
static int xalloc();

Returns: index ++.

long& iword(int idx);
```

3

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.<sup>299</sup> 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<sup>300</sup> and \*this is a base subobject of a 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.

§ 27.5.3.5 973

<sup>298)</sup> 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.

<sup>299)</sup> An implementation is free to implement both the integer array pointed at by iarray and the pointer array pointed at by parray as sparse data structures, possibly with a one-element cache for each.

300) for example, because it cannot allocate space.

```
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<sup>301</sup> 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.
- 6 Remarks: After a subsequent call to pword(int) for the same object, the earlier return value may no longer be valid.

## 27.5.3.6 ios\_base callbacks

[ios.base.callback]

void register\_callback(event\_callback fn, int index);

Effects: Registers the pair (fn,index) such that during calls to imbue() (27.5.3.3), copyfmt(), or ~ios\_base() (27.5.3.7), the function fn is called with argument index. Functions registered are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event.

2 Requires: The function fn shall not throw exceptions.

Remarks: Identical pairs are not merged. A function registered twice will be called twice.

## 27.5.3.7 ios\_base constructors/destructor

[ios.base.cons]

ios\_base();

1

2

Effects: Each ios\_base member has an indeterminate value after construction. The object's members shall be initialized by calling basic\_ios::init before the object's first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

```
~ios_base()
```

Effects: Destroys an object of class ios\_base. Calls each registered callback pair (fn, index) (27.5.3.6) as (\*fn) (erase\_event, \*this, index) at such time that any ios\_base member function called from within fn has well defined results.

## 27.5.4 Class template fpos

[fpos]

```
namespace std {
  template <class stateT> class fpos {
  public:
    // 27.5.4.1 Members
    stateT state() const;
    void state(stateT);
  private;
    stateT st; // exposition only
  };
}
```

§ 27.5.4 974

<sup>301)</sup> for example, because it cannot allocate space.

#### 27.5.4.1 fpos members

[fpos.members]

void state(stateT s);

Effects: Assign s to st.

stateT state() const;

2 Returns: Current value of st.

## 27.5.4.2 fpos requirements

[fpos.operations]

- Operations specified in Table 127 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 127 — Position type requirements

Expression	Return type	Operational semantics	$egin{array}{l} { m Assertion/note} \ { m pre-/post-condition} \end{array}$
P(i)			p == P(i)
			note: a destructor is assumed.
P p(i);			post: $p == P(i)$ .
P p = i;			
P(o)	fpos	converts from offset	
0(p)	streamoff	converts to offset	P(O(p)) == p
p == q	convertible to bool		== is an equivalence relation
p != q	convertible to bool	! (p == q)	
q = p + o	fpos	+ offset	q - o == p
p += o			
q = p - o	fpos	- offset	q + o == p
p -= o			
o = p - q	streamoff	distance	q + o == p
streamsize(o)	streamsize	converts	streamsize(O(sz)) == sz
0(sz)	streamoff	converts	streamsize(O(sz)) == sz

<sup>&</sup>lt;sup>2</sup> [Note: Every implementation is required to supply overloaded operators on fpos objects to satisfy the requirements of 27.5.4.2. It is unspecified whether these operators are members of fpos, global operators, or provided in some other way. — end note]

§ 27.5.4.2 975

<sup>3</sup> 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.

## 27.5.5 Class template basic\_ios

[ios]

27.5.5.1 Overview

[ios.overview]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_ios : public ios_base {
  public:
    // types:
    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;
    void exceptions(iostate except);
    // 27.5.5.2 Constructor/destructor:
    explicit basic_ios(basic_streambuf<charT,traits>* sb);
    virtual ~basic_ios();
    // 27.5.5.3 Members:
    basic_ostream<charT,traits>* tie() const;
    basic_ostream<charT,traits>* tie(basic_ostream<charT,traits>* tiestr);
    basic_streambuf<charT,traits>* rdbuf() const;
    basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);
    basic_ios& copyfmt(const basic_ios& rhs);
    char_type fill() const;
    char_type fill(char_type ch);
    locale imbue(const locale& loc);
             narrow(char_type c, char dfault) const;
    char_type widen(char c) const;
    basic_ios(const basic_ios& ) = delete;
    basic_ios& operator=(const basic_ios&) = delete;
 protected:
    basic_ios();
    void init(basic_streambuf<charT,traits>* sb);
```

§ 27.5.5.1 976

```
void move(basic_ios& rhs);
void move(basic_ios&& rhs);
void swap(basic_ios& rhs) noexcept;
void set_rdbuf(basic_streambuf<charT, traits>* sb);
};
};
```

#### 27.5.5.2 basic\_ios constructors

[basic.ios.cons]

explicit basic\_ios(basic\_streambuf<charT,traits>\* sb);

Effects: Constructs an object of class basic\_ios, assigning initial values to its member objects by calling init(sb).

```
basic_ios();
```

1

2 Effects: Constructs an object of class basic\_ios (27.5.3.7) leaving its member objects uninitialized. The object shall be initialized by calling basic\_ios::init before its first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

```
~basic_ios();
```

3 Remarks: The destructor does not destroy rdbuf().

```
void init(basic_streambuf<charT,traits>* sb);
```

Postconditions: The postconditions of this function are indicated in Table 128.

Table 128 — basic\_ios::init() effects

Element	Value
rdbuf()	sb
tie()	0
rdstate()	goodbit if sb is not a null pointer, otherwise
	badbit.
exceptions()	goodbit
flags()	skipws   dec
width()	0
<pre>precision()</pre>	6
fill()	widen(', ');
getloc()	a copy of the value returned by locale()
iarray	a null pointer
parray	a null pointer

## 27.5.5.3 Member functions

1

[basic.ios.members]

basic\_ostream<charT,traits>\* tie() const;

Returns: An output sequence that is *tied* to (synchronized with) the sequence controlled by the stream buffer.

§ 27.5.5.3

```
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().
   basic_streambuf<charT,traits>* rdbuf() const;
5
         Returns: A pointer to the streambuf associated with the stream.
   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().
   locale imbue(const locale& loc);
9
         Effects: Calls ios_base::imbue(loc) (27.5.3.3) and if rdbuf()!=0 then rdbuf()->pubimbue(loc)
         (27.6.3.2.1).
10
         Returns: The prior value of ios_base::imbue().
   char narrow(char_type c, char dfault) const;
         Returns: use_facet< ctype<char_type> >(getloc()).narrow(c,dfault)
   char_type widen(char c) const;
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
   § 27.5.5.3
                                                                                                      978
```

- rdstate(), rdbuf(), and exceptions() are left unchanged;
- the contents of arrays pointed at by pword and iword are copied, not the pointers themselves; 302 and
- if any newly stored pointer values in \*this point at objects stored outside the object rhs and those objects are destroyed when rhs is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects;
- 3. calls each callback pair that was copied from rhs as (\*fn)(copyfmt\_event, \*this, index);
- 4. calls exceptions(rhs.except()).
- 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 129.

Table 129 — basic\_ios::copyfmt() effects

Element	Value
rdbuf()	unchanged
tie()	rhs.tie()
rdstate()	unchanged
exceptions()	<pre>rhs.exceptions()</pre>
flags()	rhs.flags()
width()	rhs.width()
<pre>precision()</pre>	<pre>rhs.precision()</pre>
fill()	rhs.fill()
getloc()	rhs.getloc()

19 Returns: \*this.

20

```
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) noexcept;
```

21 Effects: The states of \*this and rhs shall be exchanged, except that rdbuf() shall return the same value as it returned before the function call, and rhs.rdbuf() shall return the same value as it returned before the function call.

```
void set_rdbuf(basic_streambuf<charT, traits>* sb);
```

§ 27.5.5.3 979

<sup>302)</sup> This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.

```
22
         Requires: sb != nullptr.
23
         Effects: Associates the basic_streambuf object pointed to by sb with this stream without calling
         clear().
24
         Postconditions: rdbuf() == sb.
25
         Throws: Nothing.
   27.5.5.4 basic_ios flags functions
                                                                                             [iostate.flags]
   explicit operator bool() const;
         Returns: !fail().
   bool operator!() const;
2
         Returns: fail().
   iostate rdstate() const;
3
         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).
5
         Effects: If ((state | (rdbuf() ? goodbit : badbit)) & exceptions()) == 0, returns. Oth-
         erwise, the function throws an object fail of class basic_ios::failure (27.5.3.1.1), constructed with
         implementation-defined argument values.
   void setstate(iostate state);
         Effects: Calls clear(rdstate() | state) (which may throw basic_ios::failure (27.5.3.1.1)).
   bool good() const;
7
         Returns: rdstate() == 0
   bool eof() const;
         Returns: true if eofbit is set in rdstate().
   bool fail() const;
         Returns: true if failbit or badbit is set in rdstate(). 303
   bool bad() const;
10
         Returns: true if badbit is set in rdstate().
   303) Checking badbit also for fail() is historical practice.
                                                                                                       980
   § 27.5.5.4
```

```
iostate exceptions() const;
11
         Returns: A mask that determines what elements set in rdstate() cause exceptions to be thrown.
   void exceptions(iostate except);
12
         Postcondition: except == exceptions().
13
         Effects: Calls clear(rdstate()).
             ios_base manipulators
                                                                                       [std.ios.manip]
   27.5.6
   27.5.6.1 fmtflags manipulators
                                                                                       [fmtflags.manip]
   ios_base& boolalpha(ios_base& str);
         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& noshowpoint(ios_base& str);
11
         Effects: Calls str.unsetf(ios_base::showpoint).
12
         Returns: str.
   ios_base& showpos(ios_base& str);
13
         Effects: Calls str.setf(ios_base::showpos).
14
         Returns: str.
```

§ 27.5.6.1

```
ios_base& noshowpos(ios_base& str);
15
         Effects: Calls str.unsetf(ios_base::showpos).
16
         Returns: str.
   ios_base& skipws(ios_base& str);
17
         Effects: Calls str.setf(ios_base::skipws).
         Returns: str.
   ios_base& noskipws(ios_base& str);
19
         Effects: Calls str.unsetf(ios_base::skipws).
20
         Returns: str.
   ios_base& uppercase(ios_base& str);
21
         Effects: Calls str.setf(ios_base::uppercase).
22
         Returns: str.
   ios_base& nouppercase(ios_base& str);
23
         Effects: Calls str.unsetf(ios_base::uppercase).
^{24}
         Returns: str.
   ios_base& unitbuf(ios_base& str);
^{25}
         Effects: Calls str.setf(ios_base::unitbuf).
26
         Returns: str.
   ios_base& nounitbuf(ios_base& str);
27
         Effects: Calls str.unsetf(ios_base::unitbuf).
28
         Returns: str.
                                                                                     [adjustfield.manip]
   27.5.6.2 adjustfield manipulators
   ios_base& internal(ios_base& str);
1
         Effects: Calls str.setf(ios_base::internal, ios_base::adjustfield).
         Returns: str.
   ios_base& left(ios_base& str);
3
         Effects: Calls str.setf(ios_base::left, ios_base::adjustfield).
4
         Returns: str.
   ios_base& right(ios_base& str);
5
         Effects: Calls str.setf(ios_base::right, ios_base::adjustfield).
6
         Returns: str.
   § 27.5.6.2
                                                                                                      982
```

```
[basefield.manip]
   27.5.6.3 basefield manipulators
   ios_base& dec(ios_base& str);
        Effects: Calls str.setf(ios_base::dec, ios_base::basefield).
1
        Returns: str<sup>304</sup>.
2
   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.
   27.5.6.4 floatfield manipulators
                                                                                      [floatfield.manip]
   ios_base& fixed(ios_base& str);
1
        Effects: Calls str.setf(ios_base::fixed, ios_base::floatfield).
2
        Returns: str.
   ios_base& scientific(ios_base& str);
3
        Effects: Calls str.setf(ios_base::scientific, ios_base::floatfield).
        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);
        Effects: Calls str.unsetf(ios_base::floatfield).
        Returns: str.
                    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.
```

§ 27.5.6.4 983

#### 27.5.6.5 Error reporting

[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();
```

- 3 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

[stream.buffers]

## 27.6.1 Overview

[stream.buffers.overview]

Header <streambuf> synopsis

The header <streambuf> defines types that control input from and output to character sequences.

#### 27.6.2 Stream buffer requirements

[streambuf.regts]

- 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);

§ 27.6.2 984

- the end pointer, or first element address beyond the end of the array (called **xend** here).
- <sup>3</sup> The following semantic constraints shall always apply for any set of three pointers for a sequence, using the pointer names given immediately above:
  - If xnext is not a null pointer, then xbeg and xend shall also be non-null pointers into the same charT array, as described above; otherwise, xbeg and xend shall also be null.
  - If xnext is not a null pointer and xnext < xend for an output sequence, then a write position is available. In this case, \*xnext shall be assignable as the next element to write (to put, or to store a character value, into the sequence).
  - If xnext is not a null pointer and xbeg < xnext for an input sequence, then a *putback position* is available. In this case, xnext[-1] shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.
  - If xnext is not a null pointer and xnext < xend for an input sequence, then a *read position* is available. In this case, \*xnext shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

## 27.6.3 Class template basic\_streambuf<charT,traits>

[streambuf]

```
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();
    // 27.6.3.2.1 locales:
    locale
             pubimbue(const locale& loc);
    locale
             getloc() const;
    // 27.6.3.2.2 buffer and positioning:
    basic_streambuf<char_type,traits>*
       pubsetbuf(char_type* s, streamsize n);
    pos_type pubseekoff(off_type off, ios_base::seekdir way,
      ios_base::openmode which =
          ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
      ios_base::openmode which =
          ios_base::in | ios_base::out);
    int
             pubsync();
    // Get and put areas:
    // 27.6.3.2.3 Get area:
    streamsize in_avail();
    int_type snextc();
    int_type sbumpc();
```

§ 27.6.3

```
int_type sgetc();
  streamsize sgetn(char_type* s, streamsize n);
  // 27.6.3.2.4 Putback:
  int_type sputbackc(char_type c);
  int_type sungetc();
 // 27.6.3.2.5 Put area:
 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);
 // 27.6.3.3.2 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);
  // 27.6.3.3.3 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);
 // 27.6.3.4 virtual functions:
  // 27.6.3.4.1 Locales:
 virtual void imbue(const locale& loc);
  // 27.6.3.4.2 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();
 // 27.6.3.4.3 Get area:
 virtual streamsize showmanyc();
 virtual streamsize xsgetn(char_type* s, streamsize n);
 virtual int_type underflow();
 virtual int_type
                     uflow();
  // 27.6.3.4.4 Putback:
  virtual int_type    pbackfail(int_type c = traits::eof());
 // 27.6.3.4.5 Put area:
```

§ 27.6.3

virtual streamsize xsputn(const char\_type\* s, streamsize n);

```
virtual int_type
                           overflow (int_type c = traits::eof());
     }
  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.

                                                                                        [streambuf.cons]
   27.6.3.1 basic_streambuf constructors
   basic_streambuf();
1
        Effects: Constructs an object of class basic_streambuf<charT,traits> and initializes: 305
          — all its pointer member objects to null pointers,
          — the getloc() member to a copy the global locale, locale(), at the time of construction.
2
        Remarks: Once the getloc() member is initialized, results of calling locale member functions, and of
        members of facets so obtained, can safely be cached until the next time the member imbue is called.
   basic_streambuf(const basic_streambuf& rhs);
3
        Effects: Constructs a copy of rhs.
4
        Postconditions:
          — eback() == rhs.eback()
          — gptr() == rhs.gptr()
          — egptr() == rhs.egptr()
          — pbase() == rhs.pbase()
          — pptr() == rhs.pptr()
          — epptr() == rhs.epptr()
          — getloc() == rhs.getloc()
   ~basic_streambuf();
        Effects: None.
   27.6.3.2 basic_streambuf public member functions
                                                                                   [streambuf.members]
   27.6.3.2.1 Locales
                                                                                      [streambuf.locales]
   locale pubimbue(const locale& loc);
        Postcondition: loc == getloc().
2
        Effects: Calls imbue(loc).
3
        Returns: Previous value of getloc().
   305) The default constructor is protected for class basic_streambuf to assure that only objects for classes derived from this
```

§ 27.6.3.2.1 987

class may be constructed.

```
locale getloc() const;
```

Returns: If pubimbue() has ever been called, then the last value of loc supplied, otherwise the current global locale, locale(), in effect at the time of construction. If called after pubimbue() has been called but before pubimbue has returned (i.e., from within the call of imbue()) then it returns the previous value.

```
[streambuf.buffer]
   27.6.3.2.2
               Buffer management and positioning
   basic_streambuf<char_type,traits>* pubsetbuf(char_type* s, streamsize n);
1
        Returns: setbuf(s, n).
   pos_type pubseekoff(off_type off, ios_base::seekdir way,
                  ios_base::openmode which = ios_base::in | ios_base::out);
2
        Returns: seekoff(off, way, which).
   pos_type pubseekpos(pos_type sp,
                  ios_base::openmode which = ios_base::in | ios_base::out);
3
        Returns: seekpos(sp, which).
   int pubsync();
        Returns: sync().
   27.6.3.2.3 Get area
                                                                                   [streambuf.pub.get]
   streamsize in_avail();
1
        Returns: If a read position is available, returns egptr() - gptr(). Otherwise returns showmanyc()
        (27.6.3.4.3).
   int_type snextc();
2
        Effects: Calls sbumpc().
3
        Returns: If that function returns traits::eof(), returns traits::eof(). Otherwise, returns sgetc().
   int_type sbumpc();
        Returns: If the input sequence read position is not available, returns uflow(). Otherwise, returns
        traits::to_int_type(*gptr()) and increments the next pointer for the input sequence.
   int_type sgetc();
5
        Returns: If the input sequence read position is not available, returns underflow(). Otherwise, returns
        traits::to_int_type(*gptr()).
   streamsize sgetn(char_type* s, streamsize n);
        Returns: xsgetn(s, n).
```

§ 27.6.3.2.3

```
27.6.3.2.4 Putback
                                                                              [streambuf.pub.pback]
   int_type sputbackc(char_type c);
1
        Returns: If the input sequence putback position is not available, or if traits::eq(c,gptr()[-1]) is
        false, returns pbackfail(traits::to_int_type(c)). Otherwise, decrements the next pointer for the
        input sequence and returns traits::to_int_type(*gptr()).
   int_type sungetc();
        Returns: If the input sequence putback position is not available, returns pbackfail(). Otherwise,
        decrements the next pointer for the input sequence and returns traits::to_int_type(*gptr()).
   27.6.3.2.5 Put area
                                                                                 [streambuf.pub.put]
   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);
2
        Returns: xsputn(s,n).
   27.6.3.3 basic_streambuf protected member functions
                                                                               [streambuf.protected]
   27.6.3.3.1 Assignment
                                                                                   [streambuf.assign]
   basic_streambuf& operator=(const basic_streambuf& rhs);
1
        Effects: Assigns the data members of rhs to *this.
2
        Postconditions:
         — eback() == rhs.eback()
         — gptr() == rhs.gptr()
         — egptr() == rhs.egptr()
         — pbase() == rhs.pbase()
         — pptr() == rhs.pptr()
         — epptr() == rhs.epptr()
         — getloc() == rhs.getloc()
3
        Returns: *this.
   void swap(basic_streambuf& rhs);
        Effects: Swaps the data members of rhs and *this.
```

§ 27.6.3.3.1 989

```
[streambuf.get.area]
   27.6.3.3.2 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);
        Effects: Adds n to the next pointer for the input sequence.
   void setg(char_type* gbeg, char_type* gnext, char_type* gend);
        Postconditions: gbeg == eback(), gnext == gptr(), and gend == egptr().
   27.6.3.3.3 Put area access
                                                                                    [streambuf.put.area]
   char_type* pbase() const;
        Returns: The beginning pointer for the output sequence.
   char_type* pptr() const;
        Returns: The next pointer for the output sequence.
   char_type* epptr() const;
3
        Returns: The end pointer for the output sequence.
   void pbump(int n);
        Effects: Adds n to the next pointer for the output sequence.
   void setp(char_type* pbeg, char_type* pend);
5
        Postconditions: pbeg == pbase(), pbeg == pptr(), and pend == epptr().
   27.6.3.4 basic_streambuf virtual functions
                                                                                    [streambuf.virtuals]
   27.6.3.4.1 Locales
                                                                                 [streambuf.virt.locales]
   void imbue(const locale&)
1
        Effects: Change any translations based on locale.
2
        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.
3
        Default behavior: Does nothing.
```

990

§ 27.6.3.4.1

#### 27.6.3.4.2 Buffer management and positioning

[streambuf.virt.buffer]

basic\_streambuf\* setbuf(char\_type\* s, streamsize n);

Effects: Influences stream buffering in a way that is defined separately for each class derived from basic\_streambuf in this Clause (27.8.2.4, 27.9.1.5).

2 Default behavior: Does nothing. Returns this.

- Effects: Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from basic\_streambuf in this Clause (27.8.2.4, 27.9.1.5).
- 4 Default behavior: Returns pos\_type(off\_type(-1)).

- Effects: Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from basic\_streambuf in this Clause (27.8.2, 27.9.1.1).
- 6 Default behavior: Returns pos\_type(off\_type(-1)).

int sync();

1

- 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.
- 8 Returns:-1 on failure. What constitutes failure is determined by each derived class (27.9.1.5).
- 9 Default behavior: Returns zero.

## 27.6.3.4.3 Get area

[streambuf.virt.get]

streamsize showmanyc();  $^{306}$ 

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.<sup>307</sup>

- 2 Default behavior: Returns zero.
- 3 Remarks: Uses traits::eof().

streamsize xsgetn(char\_type\* s, streamsize n);

§ 27.6.3.4.3 991

<sup>306)</sup> The morphemes of showmanyc are "es-how-many-see", not "show-manic".

<sup>307)</sup> underflow or uflow might fail by throwing an exception prematurely. The intention is not only that the calls will not return eof() but that they will return "immediately."

Effects: Assigns up to n characters to successive elements of the array whose first element is designated by s. The characters assigned are read from the input sequence as if by repeated calls to sbumpc(). Assigning stops when either n characters have been assigned or a call to sbumpc() would return traits::eof().

- 5 Returns: The number of characters assigned. 308
- 6 Remarks: Uses traits::eof().

#### int\_type underflow();

- 7 Remarks: The public members of basic\_streambuf call this virtual function only if gptr() is null or gptr() >= egptr()
- Returns: traits::to\_int\_type(c), where c is the first character of the pending sequence, without moving the input sequence position past it. If the pending sequence is null then the function returns traits::eof() to indicate failure.
- The pending sequence of characters is defined as the concatenation of:
  - a) If gptr() is non-NULL, then the egptr() gptr() characters starting at gptr(), otherwise the empty sequence.
  - b) Some sequence (possibly empty) of characters read from the input sequence.
- The result character is
  - a) If the pending sequence is non-empty, the first character of the sequence.
  - b) If the pending sequence is empty then the next character that would be read from the input sequence.
- 11 The backup sequence is defined as the concatenation of:
  - a) If eback() is null then empty,
  - b) Otherwise the gptr() eback() characters beginning at eback().
- 12 Effects: The function sets up the gptr() and egptr() satisfying one of:
  - a) If the pending sequence is non-empty, egptr() is non-null and egptr() gptr() characters starting at gptr() are the characters in the pending sequence
  - b) If the pending sequence is empty, either gptr() is null or gptr() and egptr() are set to the same non-NULL pointer.
- If eback() and gptr() are non-null then the function is not constrained as to their contents, but the "usual backup condition" is that either:
  - a) If the backup sequence contains at least gptr() eback() characters, then the gptr() eback() characters starting at eback() agree with the last gptr() eback() characters of the backup sequence.
  - b) Or the n characters starting at gptr() n agree with the backup sequence (where n is the length of the backup sequence)
- Default behavior: Returns traits::eof().

§ 27.6.3.4.3

<sup>308)</sup> Classes derived from basic\_streambuf can provide more efficient ways to implement xsgetn() and xsputn() by overriding these definitions from the base class.

```
int_type uflow();
```

Requires: The constraints are the same as for underflow(), except that the result character 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.

17 Returns: traits::eof() to indicate failure.

#### 27.6.3.4.4 Putback

[streambuf.virt.pback]

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.
- 5 Default behavior: Returns traits::eof().

#### 27.6.3.4.5 Put area

1

[streambuf.virt.put]

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.

2 Returns: The number of characters written.

```
int_type overflow(int_type c = traits::eof());
```

- 3 Effects: Consumes some initial subsequence of the characters of the pending sequence. The pending sequence is defined as the concatenation of
  - a) if pbase() is NULL then the empty sequence otherwise, pptr() pbase() characters beginning at pbase().
  - b) if traits::eq\_int\_type(c,traits::eof()) returns true, then the empty sequence otherwise, the sequence consisting of c.

§ 27.6.3.4.5

Remarks: The member functions sputc() and sputn() call this function in case that no room can be found in the put buffer enough to accommodate the argument character sequence.

- 5 Requires: Every overriding definition of this virtual function shall obey the following constraints:
  - 1) The effect of consuming a character on the associated output sequence is specified 309
  - 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.
- 6 Returns: traits::eof() or throws an exception if the function fails.

Otherwise, returns some value other than traits::eof() to indicate success. 310

7 Default behavior: Returns traits::eof().

## 27.7 Formatting and manipulators

[iostream.format]

#### 27.7.1 Overview

[iostream.format.overview]

```
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;
  typedef basic_ostream<char>
                                  ostream:
  typedef basic_ostream<wchar_t> wostream;
  template <class charT, class traits>
```

§ 27.7.1

<sup>309)</sup> That is, for each class derived from an instance of basic\_streambuf in this Clause (27.8.2, 27.9.1.1), a specification of how consuming a character effects the associated output sequence is given. There is no requirement on a program-defined class.

310) 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).

```
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);
27.7.2 Input streams
                                                                                    [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.
                                                                                             [istream]
27.7.2.1
          Class template basic_istream
  namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_istream : virtual public basic_ios<charT,traits> {
      // types (inherited from basic_ios (27.5.5)):
      typedef charT
                                         char_type;
      typedef typename traits::int_type int_type;
      typedef typename traits::pos_type pos_type;
      typedef typename traits::off_type off_type;
      typedef traits
                                         traits_type;
      // 27.7.2.1.1 Constructor/destructor:
      explicit basic_istream(basic_streambuf<charT,traits>* sb);
      virtual ~basic_istream();
      // 27.7.2.1.3 Prefix/suffix:
      class sentry;
      // 27.7.2.2 Formatted input:
      basic_istream<charT,traits>& operator>>(
        basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&));
      basic_istream<charT,traits>& operator>>(
                           basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
      basic_istream<charT,traits>& operator>>(
```

```
ios_base& (*pf)(ios_base&));
  basic_istream<charT,traits>& operator>>(bool& n);
  basic_istream<charT,traits>& operator>>(short& n);
 basic_istream<charT,traits>& operator>>(unsigned short& n);
 basic_istream<charT,traits>& operator>>(int& n);
 basic_istream<charT,traits>& operator>>(unsigned int& n);
 basic istream<charT,traits>& operator>>(long& n);
 basic_istream<charT,traits>& operator>>(unsigned long& n);
 basic_istream<charT,traits>& operator>>(long long& n);
 basic_istream<charT,traits>& operator>>(unsigned long long& n);
 basic_istream<charT,traits>& operator>>(float& f);
  basic_istream<charT,traits>& operator>>(double& f);
  basic_istream<charT,traits>& operator>>(long double& f);
 basic_istream<charT,traits>& operator>>(void*& p);
 basic_istream<charT,traits>& operator>>(
   basic_streambuf<char_type,traits>* sb);
  // 27.7.2.3 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:
  basic_istream(const basic_istream& rhs) = delete;
  basic_istream(basic_istream&& rhs);
  // 27.7.2.1.2 Assign/swap:
  basic_istream& operator=(const basic_istream& rhs) = delete;
  basic_istream& operator=(basic_istream&& rhs);
```

```
void swap(basic_istream& rhs);
  };
  // 27.7.2.2.3 character extraction templates:
  template < class charT, class traits>
    basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>&,
                                              charT&);
  template<class traits>
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>&,
                                             unsigned char&);
  template < class traits >
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>&,
                                             signed char&);
  template < class charT, class traits >
    basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>&,
                                              charT*);
  template < class traits >
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>&,
                                             unsigned char*);
  template < class traits >
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>&,
                                             signed char*);
}
```

- <sup>1</sup> The class basic\_istream defines a number of member function signatures that assist in reading and interpreting input from sequences controlled by a stream buffer.
- Two groups of member function signatures share common properties: the formatted input functions (or extractors) and the unformatted input functions. Both groups of input functions are described as if they obtain (or extract) input characters by calling rdbuf()->sbumpc() or rdbuf()->sgetc(). They may use other public members of istream.
- <sup>3</sup> If rdbuf()->sbumpc() or rdbuf()->sgetc() returns traits::eof(), then the input function, except as explicitly noted otherwise, completes its actions and does setstate(eofbit), which may throw ios\_base::failure (27.5.5.4), before returning.
- <sup>4</sup> If one of these called functions throws an exception, then unless explicitly noted otherwise, the input function sets badbit in error state. If badbit is on in exceptions(), the input function rethrows the exception without completing its actions, otherwise it does not throw anything and proceeds as if the called function had returned a failure indication.

```
27.7.2.1.1 basic_istream constructors
```

[istream.cons]

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();
```

§ 27.7.2.1.1

```
4
        Effects: Destroys an object of class basic_istream.
5
        Remarks: Does not perform any operations of rdbuf().
                                                                                         [istream.assign]
   27.7.2.1.2 Class basic_istream assign and swap
   basic_istream& operator=(basic_istream&& rhs);
1
        Effects: swap(rhs);
2
        Returns: *this.
   void swap(basic_istream& rhs);
3
        Effects: Calls basic ios<chart, traits>::swap(rhs). Exchanges the values returned by gcount()
        and rhs.gcount().
   27.7.2.1.3
               Class basic_istream::sentry
                                                                                        [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);
         explicit operator bool() const { return ok_; }
         sentry(const sentry&) = delete;
         sentry& operator=(const sentry&) = delete;
       };
     }
1
        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);
2
        Effects: If is.good() is false, calls is.setstate(failbit). Otherwise, prepares for formatted or un-
        formatted 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. 311 If noskipws is zero and
```

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.

(which may throw ios\_base::failure).

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)

To decide if the character **c** is a whitespace character, the constructor performs as if it executes the following code fragment:

§ 27.7.2.1.3 998

<sup>311)</sup> 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.

```
const ctype<charT>& ctype = use_facet<ctype<charT> >(is.getloc());
    if (ctype.is(ctype.space,c)!=0)
        // c is a whitespace character.

If, after any preparation is completed, is.good() is true, ok_ != false otherwise, ok_ == false.
    During preparation, the constructor may call setstate(failbit) (which may throw ios_base:: failure (27.5.5.4))<sup>312</sup>

-sentry();

Effects: None.

explicit operator bool() const;

Effects: Returns ok_.
```

# 27.7.2.2 Formatted input functions

[istream.formatted]

### 27.7.2.2.1 Common requirements

[istream.formatted.reqmts]

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 313 in \*this's error state. If (exceptions()&badbit) != 0 then the exception is rethrown. In any case, the formatted input function destroys the sentry object. If no exception has been thrown, it returns \*this.

#### 27.7.2.2.2 Arithmetic extractors

[istream.formatted.arithmetic]

```
operator>>(unsigned short& val);
operator>>(unsigned int& val);
operator>>(long& val);
operator>>(unsigned long& val);
operator>>(long long& val);
operator>>(unsigned long long& val);
operator>>(float& val);
operator>>(double& val);
operator>>(long double& val);
operator>>(bool& val);
operator>>(bool& val);
operator>>(void*& val);
```

1

As in the case of the inserters, these extractors depend on the locale's num\_get<> (22.4.2.1) object to perform parsing the input stream data. These extractors behave as formatted input functions (as described in 27.7.2.2.1). After a sentry object is constructed, the conversion occurs as if performed by the following code fragment:

```
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.

§ 27.7.2.2.2

<sup>312)</sup> The sentry constructor and destructor can also perform additional implementation-dependent operations.

<sup>313)</sup> This is done without causing an ios::failure to be thrown.

```
operator>>(short& val);
```

3

1

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) {</pre>
         err |= ios_base::failbit;
         val = numeric_limits<short>::max();
         else
         val = static_cast<short>(lval);
       setstate(err);
operator>>(int& val);
     The conversion occurs as if performed by the following code fragment (using the same notation as for
     the preceding code fragment):
       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) {</pre>
         err |= ios_base::failbit;
         val = numeric_limits<int>::max();
         else
         val = static_cast<int>(lval);
       setstate(err);
27.7.2.2.3
           basic_istream::operator>>
                                                                                 [istream::extractors]
basic_istream<charT,traits>& operator>>
    (basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&))
     Effects: None. This extractor does not behave as a formatted input function (as described in 27.7.2.2.1.)
```

Returns: pf(\*this).314

basic\_istream<charT,traits>& operator>>

Effects: Calls pf(\*this). This extractor does not behave as a formatted input function (as described in 27.7.2.2.1).

4 Returns: \*this.

(basic\_ios<charT,traits>& (\*pf)(basic\_ios<charT,traits>&));

§ 27.7.2.2.3

<sup>314)</sup> See, for example, the function signature ws(basic\_istream&) (27.7.2.4).

```
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 de-
         scribed in 27.7.2.2.1).
6
         Returns: *this.
    template < class charT, class traits>
      basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>& in,
                                                charT* s);
    template < class traits >
      basic_istream<char,traits>& operator>>(basic_istream<char,traits>& in,
                                               unsigned char* s);
    template < class traits >
      basic_istream<char,traits>& operator>>(basic_istream<char,traits>& in,
                                               signed char* s);
7
         Effects: Behaves like a formatted input member (as described in 27.7.2.2.1) 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()).
9
         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).
10
         If the function extracted no characters, it calls setstate(failbit), which may throw ios_base::
         failure (27.5.5.4).
11
         Returns: in.
    template < class charT, class traits>
      basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>& in,
                                                charT& c);
    template < class traits >
      basic_istream<char,traits>& operator>>(basic_istream<char,traits>& in,
                                               unsigned char& c);
    template < class traits >
      basic_istream<char,traits>& operator>>(basic_istream<char,traits>& in,
                                               signed char& c);
12
         Effects: Behaves like a formatted input member (as described in 27.7.2.2.1) of in. After a sentry
         object is constructed a character is extracted from in, if one is available, and stored in c. Otherwise,
         the function calls in.setstate(failbit).
13
         Returns: in.
    315) See, for example, the function signature dec(ios_base&) (27.5.6.3).
    § 27.7.2.2.3
                                                                                                         1001
```

```
basic_istream<charT,traits>& operator>>
  (basic_streambuf<charT,traits>* sb);
```

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). If sb is null, calls setstate(failbit), which may throw ios\_base::failure (27.5.5.4). After a sentry object is constructed, extracts characters from \*this and inserts them in the output sequence controlled by sb. Characters are extracted and inserted until any of the following occurs:

- end-of-file occurs on the input sequence;
- inserting in the output sequence fails (in which case the character to be inserted is not extracted);
- an exception occurs (in which case the exception is caught).

If the function inserts no characters, it calls setstate(failbit), which may throw ios\_base:: failure (27.5.5.4). If it inserted no characters because it caught an exception thrown while extracting characters from \*this and failbit is on in exceptions() (27.5.5.4), then the caught exception is rethrown.

16 Returns: \*this.

### 27.7.2.3 Unformatted input functions

# [istream.unformatted]

Each unformatted input function begins execution by constructing an object of class sentry with the default argument noskipws (second) argument true. If the sentry object returns true, when converted to a value of type bool, the function endeavors to obtain the requested input. Otherwise, if the sentry constructor exits by throwing an exception or if the sentry object returns false, when converted to a value of type bool, the function returns without attempting to obtain any input. In either case the number of extracted characters is set to 0; unformatted input functions taking a character array of non-zero size as an argument shall also store a null character (using charT()) in the first location of the array. If an exception is thrown during input then ios::badbit is turned on<sup>316</sup> 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;

- 2 Effects: None. This member function does not behave as an unformatted input function (as described in 27.7.2.3, paragraph 1).
- 3 Returns: The number of characters extracted by the last unformatted input member function called for the object.

```
int_type get();
```

- Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After constructing a sentry object, extracts a character c, if one is available. Otherwise, the function calls setstate(failbit), which may throw ios\_base::failure (27.5.5.4),
- 5 Returns: c if available, otherwise traits::eof().

#### basic\_istream<charT,traits>& get(char\_type& c);

<sup>316)</sup> This is done without causing an ios::failure to be thrown.

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After constructing a sentry object, extracts a character, if one is available, and assigns it to c.<sup>317</sup> Otherwise, the function calls setstate(failbit) (which may throw ios\_base::failure (27.5.5.4)).

7 Returns: \*this.

- 8 Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by s. 318 Characters are extracted and stored until any of the following occurs:
  - n is less than one or n 1 characters are stored;
  - end-of-file occurs on the input sequence (in which case the function calls setstate(eofbit));
  - traits::eq(c, delim) for the next available input character c (in which case c is not extracted).
- If the function stores no characters, it calls setstate(failbit) (which may throw ios\_base:: failure (27.5.5.4)). In any case, if n is greater than zero it then stores a null character into the next successive location of the array.
- 10 Returns: \*this.

13

```
basic_istream<charT,traits>& get(char_type* s, streamsize n)
```

- 11 Effects: Calls get(s,n,widen('\n'))
- 12 Returns: Value returned by the call.

- Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After constructing a sentry object, extracts characters and inserts them in the output sequence controlled by sb. Characters are extracted and inserted until any of the following occurs:
  - 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 (27.5.5.4).
- 15 Returns: \*this.

basic\_istream<charT,traits>& get(basic\_streambuf<char\_type,traits>& sb);

<sup>317)</sup> Note that this function is not overloaded on types signed char and unsigned char.

<sup>318)</sup> Note that this function is not overloaded on types signed char and unsigned char.

```
17 Returns: Value returned by the call.
basic_istream<charT,traits>& getline(char_type* s, streamsize n, char_type delim);
```

Effects: Calls get(sb, widen('\n'))

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by  $\mathbf{s}$ . Characters are extracted and stored until one of the following occurs:

- 1. end-of-file occurs on the input sequence (in which case the function calls setstate(eofbit));
- 2. traits::eq(c, delim) for the next available input character c (in which case the input character is extracted but not stored);<sup>320</sup>
- 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.<sup>321</sup>
- If the function extracts no characters, it calls setstate(failbit) (which may throw  $ios\_base:: failure (27.5.5.4)).$
- In any case, if n is greater than zero, it then stores a null character (using charT()) into the next successive location of the array.
- 22 Returns: \*this.
- 23 [Example:

16

18

```
#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</pre>
    else if (cin.fail()) {
      cout << "Partial long line";</pre>
      cin.clear(cin.rdstate() & ~ios_base::failbit);
    } else {
                                        // Don't include newline in count
      count--;
      cout << "Line " << ++line_number;</pre>
    cout << " (" << count << " chars): " << buffer << endl;</pre>
 }
}
```

<sup>319)</sup> Note that this function is not overloaded on types signed char and unsigned char.

<sup>320)</sup> Since the final input character is "extracted," it is counted in the gcount(), even though it is not stored.

<sup>321)</sup> This allows an input line which exactly fills the buffer, without setting failbit. This is different behavior than the historical AT&T implementation.

<sup>322)</sup> This implies an empty input line will not cause failbit to be set.

```
— end example]
    basic_istream<charT,traits>& getline(char_type* s, streamsize n);
24
         Returns: getline(s,n,widen('\n'))
    basic_istream<charT,traits>&
        ignore(streamsize n = 1, int_type delim = traits::eof());
25
         Effects: Behaves as an unformatted input function (as described in 27.7.2.3, 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() (18.3.2), 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 (27.5.5.4));
           — traits::eq_int_type(traits::to_int_type(c), delim) for the next available input charac-
              ter c (in which case c is extracted).
26
         Remarks: The last condition will never occur if traits::eq_int_type(delim, traits::eof()).
27
         Returns: *this.
    int_type peek();
28
         Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After
         constructing a sentry object, reads but does not extract the current input character.
29
         Returns: traits::eof() if good() is false. Otherwise, returns rdbuf()->sgetc().
    basic_istream<charT,traits>& read(char_type* s, streamsize n);
30
         Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After
         constructing a sentry object, if !good() calls setstate(failbit) which may throw an exception, and
         return. Otherwise extracts characters and stores them into successive locations of an array whose first
         element is designated by s.<sup>323</sup> 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 (27.5.5.4)).
31
         Returns: *this.
    streamsize readsome(char_type* s, streamsize n);
32
         Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1). After
         constructing a sentry object, if !good() calls setstate(failbit) which may throw an exception, and
         return. Otherwise extracts characters and stores them into successive locations of an array whose first
         element is designated by s. If rdbuf()->in_avail() == -1, calls setstate(eofbit) (which may
         throw ios_base::failure (27.5.5.4)), and extracts no characters;
    323) Note that this function is not overloaded on types signed char and unsigned char.
```

```
— If rdbuf()->in_avail() == 0, extracts no characters
— If rdbuf()->in_avail() > 0, extracts min(rdbuf()->in_avail(),n)).
```

Returns: The number of characters extracted.

basic\_istream<charT,traits>& putback(char\_type c);

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), except that the function first clears eofbit. After constructing a sentry object, if !good() calls setstate(failbit) which may throw an exception, and return. If rdbuf() is not null, calls rdbuf->sputbackc(). If rdbuf() is null, or if sputbackc() returns traits::eof(), calls setstate(badbit) (which may throw ios\_base::failure (27.5.5.4)). [Note: This function extracts no characters, so the value returned by the next call to gcount() is 0. — end note]

Returns: \*this.

33

35

37

38

basic\_istream<charT,traits>& unget();

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), except that the function first clears eofbit. After constructing a sentry object, if !good() calls setstate(failbit) which may throw an exception, and return. If rdbuf() is not null, calls rdbuf()->sungetc(). If rdbuf() is null, or if sungetc() returns traits::eof(), calls setstate(badbit) (which may throw ios\_base::failure (27.5.5.4)). [Note: This function extracts no characters, so the value returned by the next call to gcount() is 0. — end note]

Returns: \*this.

int sync();

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to gcount(). After constructing a sentry object, if rdbuf() is a null pointer, returns -1. Otherwise, calls rdbuf()->pubsync() and, if that function returns -1 calls setstate(badbit) (which may throw ios\_base::failure (27.5.5.4), and returns -1. Otherwise, returns zero.

pos\_type tellg();

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to gcount().

Returns: After constructing a sentry object, if fail() != false, returns pos\_type(-1) to indicate failure. Otherwise, returns rdbuf()->pubseekoff(0, cur, in).

basic\_istream<charT,traits>& seekg(pos\_type pos);

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), except that the function first clears eofbit, it does not count the number of characters extracted, and it does not affect the value returned by subsequent calls to gcount(). After constructing a sentry object, if fail() != true, executes rdbuf()->pubseekpos(pos, ios\_base::in). In case of failure, the function calls setstate(failbit) (which may throw ios\_base::failure).

42 Returns: \*this.

```
basic_istream<charT,traits>& seekg(off_type off, ios_base::seekdir dir);
```

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to gcount(). After constructing a sentry object, if fail() != true, executes rdbuf()->pubseekoff(off, dir, ios\_base::in). In case of failure, the function calls setstate(failbit) (which may throw ios\_base::failure).

44 Returns: \*this.

### 27.7.2.4 Standard basic\_istream manipulators

[istream.manip]

```
namespace std {
  template <class charT, class traits>
   basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);
}
```

Effects: Behaves as an unformatted input function (as described in 27.7.2.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to is gcount(). After constructing a sentry object extracts characters as long as the next available character c is whitespace or until there are no more characters in the sequence. Whitespace characters are distinguished with the same criterion as used by sentry::sentry (27.7.2.1.3). If we stops extracting characters because there are no more available it sets eofbit, but not failbit.

2 Returns: is.

1

### 27.7.2.5 Class template basic\_iostream

[iostreamclass]

```
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(const basic_iostream& rhs) = delete;
    basic_iostream(basic_iostream&& rhs);
    // assign/swap
    basic_iostream& operator=(const basic_iostream& rhs) = delete;
    basic_iostream& operator=(basic_iostream&& rhs);
    void swap(basic_iostream& rhs);
}
```

The class basic\_iostream inherits a number of functions that allow reading input and writing output to

```
sequences controlled by a stream buffer.
                                                                                        [iostream.cons]
   27.7.2.5.1 basic_iostream constructors
   explicit basic_iostream(basic_streambuf<charT,traits>* sb);
1
        Effects: Constructs an object of class basic_iostream, assigning initial values to the base classes
        by calling basic istream<charT, traits>(sb) (27.7.2.1) and basic ostream<charT, traits>(sb)
        (27.7.3.1)
2
        Postcondition: rdbuf()==sb and gcount()==0.
   basic_iostream(basic_iostream&& rhs);
3
        Effects: Move constructs from the rvalue rhs by constructing the basic_istream base class with
        move(rhs).
   27.7.2.5.2 basic_iostream destructor
                                                                                        [iostream.dest]
   virtual ~basic_iostream();
1
        Effects: Destroys an object of class basic_iostream.
2
        Remarks: Does not perform any operations on rdbuf().
   27.7.2.5.3 basic_iostream assign and swap
                                                                                      [iostream.assign]
   basic_iostream& operator=(basic_iostream&& rhs);
        Effects: swap(rhs).
   void swap(basic_iostream& rhs);
2
        Effects: Calls basic istream<charT, traits>::swap(rhs).
   27.7.2.6 Rvalue stream extraction
                                                                                       [istream.rvalue]
   template <class charT, class traits, class T>
     basic_istream<charT, traits>&
     operator>>(basic_istream<charT, traits>&& is, T& x);
1
        Effects: is \gg x
2
        Returns: is
   27.7.3 Output streams
                                                                                    [output.streams]
  The header <ostream> defines a type and several function signatures that control output to a stream buffer
   along with a function template that inserts into stream rvalues.
   27.7.3.1 Class template basic_ostream
                                                                                              [ostream]
     namespace std {
       template <class charT, class traits = char_traits<charT> >
       class basic_ostream : virtual public basic_ios<charT,traits> {
         // types (inherited from basic_ios (27.5.5)):
         typedef charT
                                           char_type;
         typedef typename traits::int_type int_type;
         typedef typename traits::pos_type pos_type;
   § 27.7.3.1
                                                                                                   1008
```

```
typedef typename traits::off_type off_type;
  typedef traits
                                      traits_type;
  // 27.7.3.2 Constructor/destructor:
  explicit basic_ostream(basic_streambuf<char_type,traits>* sb);
  virtual ~basic_ostream();
  // 27.7.3.4 Prefix/suffix:
  class sentry;
  // 27.7.3.6 Formatted output:
  basic_ostream<charT,traits>& operator<<(</pre>
    basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&));
  basic_ostream<charT,traits>& operator<<(</pre>
    basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
  basic_ostream<charT,traits>& operator<<(</pre>
    ios_base& (*pf)(ios_base&));
  basic_ostream<charT,traits>& operator<<(bool n);</pre>
  basic_ostream<charT,traits>& operator<<(short n);</pre>
  basic_ostream<charT,traits>& operator<<(unsigned short n);</pre>
  basic_ostream<charT,traits>& operator<<(int n);</pre>
  basic_ostream<charT,traits>& operator<<(unsigned int n);</pre>
  basic_ostream<charT,traits>& operator<<(long n);</pre>
  basic_ostream<charT,traits>& operator<<(unsigned long n);</pre>
  basic_ostream<charT,traits>& operator<<(long long n);</pre>
  basic_ostream<charT,traits>& operator<<(unsigned long long n);</pre>
  basic_ostream<charT,traits>& operator<<(float f);</pre>
  basic_ostream<charT,traits>& operator<<(double f);</pre>
  basic_ostream<charT,traits>& operator<<(long double f);</pre>
  basic_ostream<charT,traits>& operator<<(const void* p);</pre>
  basic_ostream<charT,traits>& operator<<(</pre>
    basic_streambuf<char_type,traits>* sb);
  // 27.7.3.7 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();
  // 27.7.3.5 seeks:
  pos_type tellp();
  basic_ostream<charT,traits>& seekp(pos_type);
  basic_ostream<charT,traits>& seekp(off_type, ios_base::seekdir);
protected:
  basic_ostream(const basic_ostream& rhs) = delete;
  basic_ostream(basic_ostream&& rhs);
  // 27.7.3.3 Assign/swap
  basic_ostream& operator=(basic_ostream& rhs) = delete;
  basic_ostream& operator=(const basic_ostream&& rhs);
  void swap(basic_ostream& rhs);
};
```

```
// 27.7.3.6.4 character inserters
  template < class charT, class traits>
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                             charT);
  template < class charT, class traits >
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                             char);
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                             char);
  // signed and unsigned
  template < class traits >
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                            signed char);
 template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                            unsigned char);
  template < class charT, class traits>
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                             const charT*);
  template < class charT, class traits >
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                             const char*);
  template < class traits >
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                            const char*);
  // signed and unsigned
  template < class traits >
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                            const signed char*);
  template < class traits >
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                            const unsigned char*);
}
```

- <sup>1</sup> 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.

```
27.7.3.2 basic_ostream constructors
```

[ostream.cons]

```
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) (27.5.5.2).

Postcondition: rdbuf() == sb.
```

```
virtual ~basic_ostream();
3
        Effects: Destroys an object of class basic_ostream.
4
        Remarks: Does not perform any operations on rdbuf().
   basic_ostream(basic_ostream&& rhs);
5
        Effects: Move constructs from the rvalue rhs. This is accomplished by default constructing the base
        class and calling basic_ios<charT, traits>::move(rhs) to initialize the base class.
   27.7.3.3
             Class basic_ostream assign and swap
                                                                                       [ostream.assign]
   basic_ostream& operator=(basic_ostream&& rhs);
1
        Effects: swap(rhs).
        Returns: *this.
   void swap(basic_ostream& rhs);
3
        Effects: Calls basic_ios<charT, traits>::swap(rhs).
   27.7.3.4 Class basic_ostream::sentry
                                                                                      [ostream::sentry]
     namespace std {
       template <class charT,class traits = char_traits<charT> >
       class basic_ostream<charT,traits>::sentry {
         bool ok_; // exposition only
       public:
         explicit sentry(basic_ostream<charT,traits>& os);
         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);
2
        If os.good() is nonzero, prepares for formatted or unformatted output. If os.tie() is not a null
        pointer, calls os.tie()->flush().324
3
        If, after any preparation is completed, os.good() is true, ok_ == true otherwise, ok_ == false.
        During preparation, the constructor may call setstate(failbit) (which may throw ios_base::
        failure (27.5.5.4))^{325}
   ~sentry();
4
        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 propa-
        gating an exception.
```

<sup>324)</sup> The call os.tie()->flush() does not necessarily occur if the function can determine that no synchronization is necessary.

325) The sentry constructor and destructor can also perform additional implementation-dependent operations.

```
explicit operator bool() const;
     Effects: Returns ok .
```

### 27.7.3.5 basic ostream seek members

[ostream.seeks]

Each seek member function begins execution by constructing an object of class sentry. It returns by destroying the sentry object.

```
pos_type tellp();
2
        pubseekoff(0, cur, out).
```

Returns: If fail() != false, returns pos\_type(-1) to indicate failure. Otherwise, returns rdbuf()->

basic\_ostream<charT,traits>& seekp(pos\_type pos);

- 3 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).
- 4 Returns: \*this.

```
basic_ostream<charT,traits>& seekp(off_type off, ios_base::seekdir dir);
```

- 5 Effects: If fail() != true, executes rdbuf()->pubseekoff(off, dir, ios\_base::out).
- Returns: \*this.

#### 27.7.3.6 Formatted output functions

[ostream.formatted]

# 27.7.3.6.1 Common requirements

[ostream.formatted.reqmts]

- Each formatted output function begins execution by constructing an object of class sentry. If this object returns true when converted to a value of type bool, the function endeavors to generate the requested output. If the generation fails, then the formatted output function does setstate(ios base::failbit), which might throw an exception. If an exception is thrown during output, then ios::badbit is turned on 326 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.

#### 27.7.3.6.2 Arithmetic inserters

[ostream.inserters.arithmetic]

```
operator<<(bool val);</pre>
operator << (short val);
operator << (unsigned short val);
operator<<(int val);
operator << (unsigned int val);
operator << (long val);
operator << (unsigned long val);
operator << (long long val);
operator<<(unsigned long long val);</pre>
operator<<(float val);
operator << (double val);
operator << (long double val);
operator<<(const void* val);</pre>
```

326) without causing an ios::failure to be thrown.

§ 27.7.3.6.2 1012

Effects: The classes num\_get<> and num\_put<> handle locale-dependent numeric formatting and parsing. These inserter functions use the imbued locale value to perform numeric formatting. When val is of type bool, long, unsigned long, long long, unsigned long, double, long double, or const void\*, the formatting conversion occurs as if it performed the following code fragment:

When val is of type short the formatting conversion occurs as if it performed the following code fragment:

```
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:

```
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
   num_put<charT,ostreambuf_iterator<charT,traits> >
   >(getloc()).put(*this, *this, fill(),
   baseflags == ios_base::oct || baseflags == ios_base::hex
    ? static_cast<long>(static_cast<unsigned int>(val))
    : static_cast<long>(val)).failed();
```

When val is of type unsigned short or unsigned int the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
       static_cast<unsigned long>(val)).failed();
```

When val is of type float the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
       static_cast<double>(va})).failed();
```

The first argument provides an object of the ostreambuf\_iterator<> class which is an iterator for class basic\_ostream<>. It bypasses ostreams and uses streambufs directly. Class locale relies on these types as its interface to iostreams, since for flexibility it has been abstracted away from direct dependence on ostream. The second parameter is a reference to the base subobject of type ios\_base. It provides formatting specifications such as field width, and a locale from which to obtain other facets. If failed is true then does setstate(badbit), which may throw an exception, and returns.

3 Returns: \*this.

§ 27.7.3.6.2

```
27.7.3.6.3 basic ostream::operator<<
                                                                                        [ostream.inserters]
    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 27.7.3.6.1).
         Returns: pf(*this).327
2
    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 27.7.3.6.1).
         Returns: *this.328
4
    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 27.7.3.6.1).
6
         Returns: *this.
    basic_ostream<charT,traits>& operator<<
        (basic_streambuf<charT,traits>* sb);
7
         Effects: Behaves as an unformatted output function (as described in 27.7.3.7, paragraph 1). Af-
         ter 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 (27.5.5.4)). If an exception was thrown while extracting a character, the function sets failbit
         in error state, and if failbit is on in exceptions() the caught exception is rethrown.
10
         Returns: *this.
    27.7.3.6.4 Character inserter function templates
                                                                             [ostream.inserters.character]
    template < class charT, class traits>
      basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
                                                charT c};
    template < class charT, class traits>
      basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
                                                char c):
    327) See, for example, the function signature endl(basic_ostream&) (27.7.3.8).
    328) See, for example, the function signature dec(ios_base&) (27.5.6.3).
    § 27.7.3.6.4
                                                                                                        1014
```

```
// specialization
template < class traits >
 basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                          char c):
 // signed and unsigned
template < class traits >
 basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                          signed char c);
template < class traits >
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                          unsigned char c);
     Effects: Behaves like a formatted inserter (as described in 27.7.3.6.1) 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 27.7.3.6.1) of out. After a sentry object is
     constructed it inserts n characters starting at s, where n is the number that would be computed as if
     by:
       — traits::length(s) for the overload where the first argument is of type basic ostream<chart,
          traits>& and the second is of type const charT*, and also for the overload where the first
          argument is of type basic_ostream<char, traits>& and the second is of type const char*,
          std::char_traits<char>::length(s) for the overload where the first argument is of type
          basic_ostream<charT, traits>& and the second is of type const char*,
      — traits::length(reinterpret_cast<const char*>(s)) for the other two overloads.
     Padding is determined as described in 22.4.2.2.2. The n characters starting at s are widened using
     out.widen (27.5.5.3). The widened characters and any required padding are inserted into out. Calls
     width(0).
```

2

3

4

5

6

Returns: out.

§ 27.7.3.6.4

### 27.7.3.7 Unformatted output functions

### [ostream.unformatted]

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<sup>329</sup> in \*this's error state. If (exceptions() & badbit) != 0 then the exception is rethrown. In any case, the unformatted output function ends by destroying the sentry object, then, if no exception was thrown, returning the value specified for the unformatted output function.

basic\_ostream<charT,traits>& put(char\_type c);

- Effects: Behaves as an unformatted output function (as described in 27.7.3.7, paragraph 1). After constructing a sentry object, inserts the character c, if possible.<sup>330</sup>
- Otherwise, calls setstate(badbit) (which may throw ios base::failure (27.5.5.4)).
- 4 Returns: \*this.

basic\_ostream& write(const char\_type\* s, streamsize n);

- Effects: Behaves as an unformatted output function (as described in 27.7.3.7, paragraph 1). After constructing a sentry object, obtains characters to insert from successive locations of an array whose first element is designated by s.<sup>331</sup> 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 (27.5.5.4)).
- 6 Returns: \*this.

basic\_ostream& flush();

- Effects: Behaves as an unformatted output function (as described in 27.7.3.6.1, paragraph 1). If rdbuf() is not a null pointer, constructs a sentry object. If this object returns true when converted to a value of type bool the function calls rdbuf()->pubsync(). If that function returns -1 calls setstate(badbit) (which may throw ios\_base::failure (27.5.5.4)). Otherwise, if the sentry object returns false, does nothing.
- 8 Returns: \*this.

1

2

# 27.7.3.8 Standard basic\_ostream manipulators

[ostream.manip]

```
namespace std {
  template <class charT, class traits>
    basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);
}

Effects: Calls os.put(os.widen('\n')), then os.flush().

Returns: os.
```

<sup>329)</sup> without causing an ios::failure to be thrown.

<sup>330)</sup> Note that this function is not overloaded on types signed char and unsigned char.

<sup>331)</sup> Note that this function is not overloaded on types signed char and unsigned char.

```
namespace std {
     template <class charT, class traits>
       basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>& os);
   }
3
        Effects: Inserts a null character into the output sequence: calls os.put(charT()).
4
        Returns: os.
   namespace std {
     template <class charT, class traits>
       basic_ostream<charT,traits>& flush(basic_ostream<charT,traits>& os);
5
        Effects: Calls os.flush().
6
        Returns: os.
   27.7.3.9
            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);
1
        Effects: os \ll x
2
        Returns: os
   27.7.4 Standard manipulators
                                                                                             [std.manip]
  The header <iomanip> defines several functions that support extractors and inserters that alter information
   maintained by class ios_base and its derived classes.
   unspecified resetiosflags(ios_base::fmtflags mask);
2
        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:332
          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);

3

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:

<sup>332)</sup> 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 <<restiosflags(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).

```
void f(ios_base& str, ios_base::fmtflags mask) {
   // set specified flags
   str.setf(mask);
}
```

The expression out << setiosflags(mask) shall have type basic\_ostream<charT, traits>& and value out. The expression in >> setiosflags(mask) shall have type basic\_istream<charT, traits>& and value in.

unspecified setbase(int base);

Returns: An object of unspecified type such that if out is an object of type basic\_ostream<charT, traits> then the expression out << setbase(base) behaves as if it called f(out, base), or if in is an object of type basic\_istream<charT, traits> then the expression in >> setbase(base) behaves as if it called f(in, base), where the function f is defined as:

```
void f(ios_base& str, int base) {
    // set basefield
    str.setf(base == 8 ? ios_base::oct :
        base == 10 ? ios_base::dec :
        base == 16 ? ios_base::hex :
        ios_base::fmtflags(0), ios_base::basefield);
}
```

The expression out << setbase(base) shall have type basic\_ostream<charT, traits>& and value out. The expression in >> setbase(base) shall have type basic\_istream<charT, traits>& and value in.

```
unspecified setfill(char_type c);
```

Returns: An object of unspecified type such that if out is an object of type basic\_ostream<charT, traits> and c has type charT then the expression out << setfill(c) behaves as if it called f(out, c), where the function f is defined as:

```
template<class charT, class traits>
void f(basic_ios<charT,traits>& str, charT c) {
   // set fill character
   str.fill(c);
}
```

The expression out << setfill(c) shall have type basic\_ostream<charT, traits>& and value out.

```
unspecified setprecision(int n);
```

Returns: An object of unspecified type such that if out is an object of type basic\_ostream<chart, traits> then the expression out << setprecision(n) behaves as if it called f(out, n), or if in is an object of type basic\_istream<chart, traits> then the expression in >> setprecision(n) behaves as if it called f(in, n), where the function f is defined as:

```
void f(ios_base& str, int n) {
   // set precision
   str.precision(n);
}
```

The expression out << setprecision(n) shall have type basic\_ostream<charT, traits>& and value out. The expression in >> setprecision(n) shall have type basic\_istream<charT, traits>& and value in.

```
unspecified setw(int n);
```

Returns: An object of unspecified type such that if out is an instance of basic\_ostream<charT, traits> then the expression out << setw(n) behaves as if it called f(out, n), or if in is an object of type basic\_istream<charT, traits> then the expression in >> setw(n) behaves as if it called f(in, n), where the function f is defined as:

```
void f(ios_base& str, int n) {
   // set width
   str.width(n);
}
```

The expression out << setw(n) shall have type basic\_ostream<charT, traits>& and value out. The expression in >> setw(n) shall have type basic\_istream<charT, traits>& and value in.

# 27.7.5 Extended manipulators

[ext.manip]

The header **<iomanip>** defines several functions that support extractors and inserters that allow for the parsing and formatting of sequences and values for money and time.

template <class moneyT> unspecified get\_money(moneyT& mon, bool intl = false);

- Requires: The type moneyT shall be either long double or a specialization of the basic\_string template (Clause 21).
- Effects: The expression in » get\_money(mon, intl) described below behaves as a formatted input function (27.7.2.2.1).
- Returns: An object of unspecified type such that if in is an object of type basic\_istream<charT, traits> then the expression in >> get\_money(mon, intl) behaves as if it called f(in, mon, intl), where the function f is defined as:

```
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);

Requires: The type moneyT shall be either long double or a specialization of the basic\_string template (Clause 21).

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:

```
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_ios<charT, traits>& str, struct tm* tmb, const charT* fmt) {
  typedef istreambuf_iterator<charT, traits> Iter;
  typedef time_get<charT, Iter> TimeGet;

  ios_base::iostate err = ios_base::goodbit;
  const TimeGet& tg = use_facet<TimeGet>(str.getloc());

  tg.get(Iter(str.rdbuf()), Iter(), str, err, tmb,
    fmt, fmt + traits::length(fmt));

  if (err != ios_base::goodbit)
    str.setstate(err):
}
```

The expression in >> get\_time(tmb, fmt) shall have type basic\_istream<charT, traits>& and value in.

template <class charT> unspecified put\_time(const struct tm\* tmb, const charT\* fmt);

Requires: The argument tmb shall be a valid pointer to an object of type struct tm, and the argument fmt shall be a valid pointer to an array of objects of type charT with char\_traits<charT>::length(fmt) elements.

Returns: An object of unspecified type such that if out is an object of type basic\_ostream<charT, traits> then the expression out << put\_time(tmb, fmt) behaves as if it called f(out, tmb, fmt), where the function f is defined as:

```
template <class charT, class traits>
void f(basic_ios<charT, traits>& str, const struct tm* tmb, const charT* fmt) {
  typedef ostreambuf_iterator<charT, traits> Iter;
  typedef time_put<charT, Iter> TimePut;

  const TimePut& tp = use_facet<TimePut>(str.getloc());
  const Iter end = tp.put(Iter(str.rdbuf()), str, str.fill(), tmb,
      fmt, fmt + traits::length(fmt));

  if (end.failed())
      str.setstate(ios_base::badbit);
}
```

The expression out << put\_time(tmb, fmt) shall have type basic\_istream<charT, traits>& and value out.

# 27.8 String-based streams

[string.streams]

### 27.8.1 Overview

[string.streams.overview]

The header <sstream> defines four class templates and eight types that associate stream buffers with objects of class basic\_string, as described in 21.3.

```
Header <sstream> synopsis
```

```
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;
                                       stringstream;
 typedef basic_stringstream<char>
 typedef basic_stringstream<wchar_t> wstringstream;
```

# 27.8.2 Class template basic\_stringbuf

[stringbuf]

§ 27.8.2

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
  class basic_stringbuf : public basic_streambuf<charT,traits> {
  public:
    typedef charT
                                       char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits
                                      traits_type;
    typedef Allocator
                                      allocator_type;
    // 27.8.2.1 Constructors:
    explicit basic_stringbuf(ios_base::openmode which
                               = ios_base::in | ios_base::out);
    explicit basic_stringbuf
    (const basic_string<charT,traits,Allocator>& str,
     ios_base::openmode which = ios_base::in | ios_base::out);
    basic_stringbuf(const basic_stringbuf& rhs) = delete;
    basic_stringbuf(basic_stringbuf&& rhs);
    // 27.8.2.2 Assign and swap:
    basic_stringbuf& operator=(const basic_stringbuf& rhs) = delete;
    basic_stringbuf& operator=(basic_stringbuf&& rhs);
    void swap(basic_stringbuf& rhs);
    // 27.8.2.3 Get and set:
    basic_string<charT,traits,Allocator> str() const;
    void str(const basic_string<charT,traits,Allocator>& s);
 protected:
    // 27.8.2.4 Overridden virtual functions:
    virtual int_type underflow();
    virtual int_type    pbackfail(int_type c = traits::eof());
    virtual int_type overflow (int_type c = traits::eof());
    virtual basic_streambuf<charT,traits>* setbuf(charT*, streamsize);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);
  private:
    ios_base::openmode mode; // exposition only
  };
  template <class charT, class traits, class Allocator>
  void swap(basic_stringbuf<charT, traits, Allocator>& x,
            basic_stringbuf<charT, traits, Allocator>& y);
```

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,

§ 27.8.2

or made available as, an object of class basic\_string.

- For the sake of exposition, the maintained data is presented here as:
  - ios\_base::openmode mode, has in set if the input sequence can be read, and out set if the output sequence can be written.

#### 27.8.2.1 basic\_stringbuf constructors

[stringbuf.cons]

- Effects: Constructs an object of class basic\_stringbuf, initializing the base class with basic\_streambuf() (27.6.3.1), and initializing mode with which.
- Postcondition: str() == "".

Effects: Constructs an object of class basic\_stringbuf, initializing the base class with basic\_streambuf() (27.6.3.1), and initializing mode with which. Then calls str(s).

basic\_stringbuf(basic\_stringbuf&& rhs);

- Effects: Move constructs from the rvalue rhs. It is implementation-defined whether the sequence pointers in \*this (eback(), gptr(), egptr(), pbase(), pptr(), epptr()) obtain the values which rhs had. Whether they do or not, \*this and rhs reference separate buffers (if any at all) after the construction. The openmode, locale and any other state of rhs is also copied.
- Postconditions: Let rhs\_p refer to the state of rhs just prior to this construction and let rhs\_a refer to the state of rhs just after this construction.

```
- str() == rhs_p.str()
- gptr() - eback() == rhs_p.gptr() - rhs_p.eback()
- egptr() - eback() == rhs_p.egptr() - rhs_p.eback()
- pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()
- 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()) epptr() != rhs_a.pptr()
- if (epptr()) epptr() != rhs_a.epptr()
```

### 27.8.2.2 Assign and swap

[stringbuf.assign]

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 27.8.2.1).

2 Returns: \*this.

§ 27.8.2.2

```
void swap(basic_stringbuf& rhs);
```

Effects: Exchanges the state of \*this and rhs.

4 Effects: x.swap(y).

3

1

1

#### 27.8.2.3 Member functions

[stringbuf.members]

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);

- 2 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 epptr() >= pbase() + s.size() holds; in addition, if mode & ios\_base::ate is true, pptr() == pbase() + s.size() holds, otherwise pptr() == pbase() is true. If mode & ios\_base::in is true, eback() points to the first underlying character, and both gptr() == eback() and egptr() == eback() + s.size() hold.

# 27.8.2.4 Overridden virtual functions

[stringbuf.virtuals]

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 put-back position available, and if traits::eq(to\_char\_type(c),gptr()[-1]) returns true, assigns gptr() 1 to gptr().

Returns: c.

§ 27.8.2.4

```
— If traits::eq_int_type(c,traits::eof()) returns false and if the input sequence has a put-back position available, and if mode & ios_base::out is nonzero, assigns c to *--gptr().
Returns: c.
```

If traits::eq\_int\_type(c,traits::eof()) returns true and if the input sequence has a put-back position available, assigns gptr() - 1 to gptr().

Returns: traits::not\_eof(c).

- 3 Returns: 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.

```
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 traits::eq\_int\_type(c,traits::eof()) returns false and if either the output sequence has a write position available or the function makes a write position available (as described below), the function calls sputc(c).
    - Signals success by returning c.
  - If traits::eq\_int\_type(c,traits::eof()) returns true, there is no character to append. Signals success by returning a value other than traits::eof().
- 6 Remarks: The function can alter the number of write positions available as a result of any call.
- 7 Returns: traits::eof() to indicate failure.
- The function can make a write position available only if (mode & ios\_base::out) != 0. To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus at least one additional write position. If (mode & ios\_base::in) != 0, the function alters the read end pointer egptr() to point just past the new write position.

- 9 Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 130.
- For a sequence to be positioned, if its next pointer (either gptr() or pptr()) is a null pointer and the new offset newoff is nonzero, the positioning operation fails. Otherwise, the function determines newoff as indicated in Table 131.
- If (newoff + off) < 0, or if newoff + off refers to an uninitialized character (as defined in 27.8.2.3 paragraph 1), the positioning operation fails. Otherwise, the function assigns xbeg + newoff + off to the next pointer xnext.
- Returns: pos\_type(newoff), constructed from the resultant offset newoff (of type off\_type), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is pos\_type(off\_type(-1)).

§ 27.8.2.4

Table 130 — seekoff positioning

Conditions	Result
(which & ios_base::in) == ios	positions the input sequence
base::in	
(which & ios_base::out) == ios	positions the output sequence
base::out	
(which & (ios_base::in	positions both the input and the output sequences
ios_base::out)) ==	
(ios_base::in)	
ios_base::out))	
and way == either	
ios_base::beg or	
ios_base::end	
Otherwise	the positioning operation fails.

Table 131 — newoff values

Condition	newoff Value
way == ios_base::beg	0
way == ios_base::cur	the next pointer minus the begin-
	ning pointer (xnext - xbeg).
way == ios_base::end	the high mark pointer minus the
	beginning pointer (high_mark -
	xbeg).

```
Effects: Equivalent to seekoff(off_type(sp), ios_base::beg, which).
```

Returns: sp to indicate success, or pos\_type(off\_type(-1)) to indicate failure.

```
basic_streambuf<charT,traits>* setbuf(charT* s, streamsize n);
```

- Effects: implementation-defined, except that setbuf(0,0) has no effect.
- 16 Returns: this.

### 27.8.3 Class template basic\_istringstream

[istringstream]

§ 27.8.3

```
// 27.8.3.1 Constructors:
         explicit basic_istringstream(ios_base::openmode which = ios_base::in);
         explicit basic_istringstream(
                    const basic_string<charT,traits,Allocator>& str,
                    ios_base::openmode which = ios_base::in);
         basic_istringstream(const basic_istringstream& rhs) = delete;
         basic_istringstream(basic_istringstream&& rhs);
         // 27.8.3.2 Assign and swap:
         basic_istringstream& operator=(const basic_istringstream& rhs) = delete;
         basic_istringstream& operator=(basic_istringstream&& rhs);
         void swap(basic_istringstream& rhs);
         // 27.8.3.3 Members:
         basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
         basic_string<charT,traits,Allocator> str() const;
         void str(const basic_string<charT,traits,Allocator>& s);
         basic_stringbuf<charT,traits,Allocator> sb; // exposition only
       };
       template <class charT, class traits, class Allocator>
       void swap(basic_istringstream<charT, traits, Allocator>& x,
                 basic_istringstream<charT, traits, Allocator>& y);
     }
  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.
   27.8.3.1 basic_istringstream constructors
                                                                                  [istringstream.cons]
   explicit basic_istringstream(ios_base::openmode which = ios_base::in);
1
        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, Alloca-
        tor>(which | ios base::in)) (27.8.2.1).
   explicit basic_istringstream(
                   const basic_string<charT, traits, Allocator>& str,
                   ios_base::openmode which = ios_base::in);
        Effects: Constructs an object of class basic_istringstream<charT, traits>, initializing the base
        class with basic_istream(&sb) and initializing sb with basic_stringbuf<charT, traits, Alloca-
        tor>(str, which | ios_base::in)) (27.8.2.1).
   basic_istringstream(basic_istringstream&& rhs);
3
        Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base
        class, and the contained basic_stringbuf. Next basic_istream<charT,traits>::set_rdbuf(&sb)
        is called to install the contained basic_stringbuf.
```

§ 27.8.3.1

[istringstream.assign]

1028

27.8.3.2 Assign and swap

§ 27.8.4

```
basic_istringstream& operator=(basic_istringstream&& rhs);
1
        Effects: Move assigns the base and members of *this from the base and corresponding members of
        rhs.
        Returns: *this.
   void swap(basic_istringstream& rhs);
3
        Effects: Exchanges the state of *this and rhs by calling basic_istream<charT, traits>::swap(rhs)
        and sb.swap(rhs.sb).
   template <class charT, class traits, class Allocator>
   void swap(basic_istringstream<charT, traits, Allocator>& x,
             basic_istringstream<charT, traits, Allocator>& y);
4
        Effects: x.swap(y).
   27.8.3.3 Member functions
                                                                             [istringstream.members]
   basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
        Returns: const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb).
   basic_string<charT,traits,Allocator> str() const;
        Returns: rdbuf()->str().
   void str(const basic_string<charT,traits,Allocator>& s);
3
        Effects: Calls rdbuf()->str(s).
                                                                                     [ostringstream]
   27.8.4 Class template basic_ostringstream
     namespace std {
       template <class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT> >
       class basic_ostringstream : public basic_ostream<charT,traits> {
       public:
        // types:
         typedef charT
                                           char_type;
         typedef typename traits::int_type int_type;
         typedef typename traits::pos_type pos_type;
         typedef typename traits::off_type off_type;
         typedef traits
                                           traits_type;
         typedef Allocator
                                           allocator_type;
         // 27.8.4.1 Constructors/destructor:
         explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
         explicit basic_ostringstream(
                  const basic_string<charT,traits,Allocator>& str,
                  ios_base::openmode which = ios_base::out);
```

```
basic ostringstream(const basic ostringstream& rhs) = delete;
         basic_ostringstream(basic_ostringstream&& rhs);
         // 27.8.4.2 Assign/swap:
         basic_ostringstream& operator=(const basic_ostringstream& rhs) = delete;
         basic_ostringstream& operator=(basic_ostringstream&& rhs);
         void swap(basic_ostringstream& rhs);
         // 27.8.4.3 Members:
         basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
         basic_string<charT,traits,Allocator> str() const;
         void
                 str(const basic_string<charT,traits,Allocator>& s);
        private:
         basic_stringbuf<charT,traits,Allocator> sb; // exposition only
       };
       template <class charT, class traits, class Allocator>
       void swap(basic_ostringstream<charT, traits, Allocator>& x,
                 basic_ostringstream<charT, traits, Allocator>& y);
     }
  The class basic_ostringstream<charT, traits, Allocator> supports writing objects of class basic_-
   string<charT, traits, Allocator>. It uses a basic_stringbuf object to control the associated storage.
   For the sake of exposition, the maintained data is presented here as:
     — sb, the stringbuf object.
                                                                                  [ostringstream.cons]
   27.8.4.1 basic_ostringstream constructors
   explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
        Effects: Constructs an object of class basic_ostringstream, initializing the base class with basic_-
        ostream(&sb) and initializing sb with basic_stringbuf<charT, traits, Allocator>(which |
        ios_base::out)) (27.8.2.1).
   explicit basic_ostringstream(
     const basic_string<charT,traits,Allocator>& str,
     ios_base::openmode which = ios_base::out);
        Effects: Constructs an object of class basic_ostringstream<charT, traits>, initializing the base
        class with basic_ostream(&sb) and initializing sb with basic_stringbuf<charT, traits, Alloca-
        tor>(str, which | ios_base::out)) (27.8.2.1).
   basic_ostringstream(basic_ostringstream&& rhs);
3
        Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base
        class, and the contained basic_stringbuf. Next basic_ostream<charT,traits>::set_rdbuf(&sb)
        is called to install the contained basic_stringbuf.
   27.8.4.2
             Assign and swap
                                                                                [ostringstream.assign]
   basic_ostringstream& operator=(basic_ostringstream&& rhs);
        Effects: Move assigns the base and members of *this from the base and corresponding members of
        rhs.
        Returns: *this.
   § 27.8.4.2
                                                                                                   1029
```

2

1

```
void swap(basic_ostringstream& rhs);
3
        Effects: Exchanges the state of *this and rhs by calling basic_ostream<charT, traits>::swap(rhs)
        and sb.swap(rhs.sb).
   template <class charT, class traits, class Allocator>
   void swap(basic_ostringstream<charT, traits, Allocator>& x,
             basic_ostringstream<charT, traits, Allocator>& y);
        Effects: x.swap(y).
   27.8.4.3 Member functions
                                                                            [ostringstream.members]
   basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
        Returns: const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb).
   basic_string<charT,traits,Allocator> str() const;
        Returns: rdbuf()->str().
   void str(const basic_string<charT,traits,Allocator>& s);
        Effects: Calls rdbuf()->str(s).
   27.8.5 Class template basic_stringstream
                                                                                      [stringstream]
    namespace std {
       template <class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT> >
       class basic_stringstream
         : public basic_iostream<charT,traits> {
       public:
         // types:
         typedef charT
                                           char_type;
         typedef typename traits::int_type int_type;
         typedef typename traits::pos_type pos_type;
         typedef typename traits::off_type off_type;
         typedef traits
                                           traits_type;
         typedef Allocator
                                           allocator_type;
         // constructors/destructor
         explicit basic_stringstream(
         ios_base::openmode which = ios_base::out|ios_base::in);
         explicit basic_stringstream(
         const basic_string<charT,traits,Allocator>& str,
         ios_base::openmode which = ios_base::out|ios_base::in);
         basic_stringstream(const basic_stringstream& rhs) = delete;
         basic_stringstream(basic_stringstream&& rhs);
         // 27.8.6.1 Assign/swap:
         basic_stringstream& operator=(const basic_stringstream& rhs) = delete;
        basic_stringstream& operator=(basic_stringstream&& rhs);
```

§ 27.8.5

```
void swap(basic_stringstream& rhs);
         // Members:
         basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
        basic_string<charT,traits,Allocator> str() const;
         void str(const basic_string<charT,traits,Allocator>& str);
      private:
        basic_stringbuf<charT, traits> sb; // exposition only
       template <class charT, class traits, class Allocator>
       void swap(basic_stringstream<charT, traits, Allocator>& x,
                 basic_stringstream<charT, traits, Allocator>& y);
     }
  The class template basic_stringstream<charT, traits> supports reading and writing from objects of
   class basic_string<charT, traits, Allocator>. It uses a basic_stringbuf<charT, traits, Alloca-
   tor> object to control the associated sequence. For the sake of exposition, the maintained data is presented
   here as
     — sb, the stringbuf object.
            basic_stringstream constructors
                                                                                 [stringstream.cons]
   explicit basic_stringstream(
       ios_base::openmode which = ios_base::out|ios_base::in);
        Effects: Constructs an object of class basic_stringstream<chart,traits>, initializing the base
        class with basic_iostream(&sb) and initializing sb with basic_stringbuf<charT,traits,Alloca-
        tor>(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, Alloca-
        tor>(str, which).
   basic_stringstream(basic_stringstream&& rhs);
3
        Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base
        class, and the contained basic_stringbuf. Next basic_istream<charT,traits>::set_rdbuf(&sb)
        is called to install the contained basic_stringbuf.
   27.8.6.1 Assign and swap
                                                                                 [stringstream.assign]
   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);
   § 27.8.6.1
                                                                                                  1031
```

2

1

```
3
        Effects: Exchanges the state of *this and rhs by calling basic_iostream<charT, traits>::swap(rhs)
        and sb.swap(rhs.sb).
   template <class charT, class traits, class Allocator>
   void swap(basic_stringstream<charT, traits, Allocator>& x,
             basic_stringstream<charT, traits, Allocator>& y);
        Effects: x.swap(y).
   27.8.7 Member functions
                                                                           [stringstream.members]
   basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
1
        Returns: const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb)
   basic_string<charT,traits,Allocator> str() const;
        Returns: rdbuf()->str().
   void str(const basic_string<charT,traits,Allocator>& str);
        Effects: Calls rdbuf()->str(str).
   27.9
        File-based streams
                                                                                        [file.streams]
   27.9.1 File streams
                                                                                           [fstreams]
  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;
```

§ 27.9.1

- <sup>2</sup> 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

#### 27.9.1.1 Class template basic\_filebuf

[filebuf]

```
namespace std {
 template <class charT, class traits = char_traits<charT> >
 class basic_filebuf : public basic_streambuf<charT,traits> {
   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(const basic_filebuf& rhs) = delete;
    basic_filebuf(basic_filebuf&& rhs);
    virtual ~basic_filebuf();
    // 27.9.1.3 Assign/swap:
    basic_filebuf& operator=(const basic_filebuf& rhs) = delete;
    basic_filebuf& operator=(basic_filebuf&& rhs);
    void swap(basic_filebuf& rhs);
    // 27.9.1.4 Members:
   bool is_open() const;
   basic_filebuf<charT,traits>* open(const char* s,
        ios_base::openmode mode);
    basic_filebuf<charT,traits>* open(const string& s,
        ios_base::openmode mode);
    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);
}
```

The class basic\_filebuf<charT, traits> associates both the input sequence and the output sequence with a file.

- The restrictions on reading and writing a sequence controlled by an object of class basic\_filebuf<charT, traits> are the same as for reading and writing with the Standard C library FILEs.
- 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.
- 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.
- <sup>5</sup> 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

```
const codecvt<charT,char,typename traits::state_type>& a_codecvt =
  use_facet<codecvt<charT,char,typename traits::state_type> >(getloc());
```

#### 27.9.1.2 basic\_filebuf constructors

[filebuf.cons]

basic\_filebuf();

1

Effects: Constructs an object of class basic\_filebuf<charT,traits>, initializing the base class with basic streambuf<charT,traits>() (27.6.3.1).

Postcondition: is\_open() == false.

```
basic_filebuf(basic_filebuf&& rhs);
```

- Effects: Move constructs from the rvalue rhs. It is implementation-defined whether the sequence pointers in \*this (eback(), gptr(), egptr(), pbase(), pptr(), epptr()) obtain the values which rhs had. Whether they do or not, \*this and rhs reference separate buffers (if any at all) after the construction. Additionally \*this references the file which rhs did before the construction, and rhs references no file after the construction. The openmode, locale and any other state of rhs is also copied.
- Postconditions: Let rhs\_p refer to the state of rhs just prior to this construction and let rhs\_a refer to the state of rhs just after this construction.

```
- is_open() == rhs_p.is_open()
- rhs_a.is_open() == false
- gptr() - eback() == rhs_p.gptr() - rhs_p.eback()
- egptr() - eback() == rhs_p.egptr() - rhs_p.eback()
- pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()
- epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()
- if (eback()) eback() != rhs_a.eback()
- if (gptr()) gptr() != rhs_a.gptr()
- if (egptr()) egptr() != rhs_a.egptr()
```

— if (pbase()) pbase() != rhs\_a.pbase()

```
— if (pptr()) pptr() != rhs_a.pptr()
          — if (epptr()) epptr() != rhs_a.epptr()
   virtual ~basic_filebuf();
5
         Effects: Destroys an object of class basic_filebuf<charT, traits>. Calls close(). If an exception
        occurs during the destruction of the object, including the call to close(), the exception is caught but
        not rethrown (see 17.6.5.12).
   27.9.1.3
             Assign and swap
                                                                                            [filebuf.assign]
   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).
2
         Returns: *this.
   void swap(basic_filebuf& rhs);
3
         Effects: Exchanges the state of *this and rhs.
   template <class charT, class traits>
   void swap(basic_filebuf<charT, traits>& x,
             basic_filebuf<charT, traits>& y);
         Effects: x.swap(y).
   27.9.1.4 Member functions
                                                                                         [filebuf.members]
   bool is_open() const;
        Returns: true if a previous call to open succeeded (returned a non-null value) and there has been no
        intervening call to close.
   basic_filebuf<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.
2
        It then opens a file, if possible, whose name is the NTBS s (as if by calling std::fopen(s,modstr)).
        The NTBS modstr is determined from mode & ~ios_base::ate as indicated in Table 132. If mode is
        not some combination of flags shown in the table then the open fails.
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)).333
4
        If the repositioning operation fails, calls close() and returns a null pointer to indicate failure.
5
         Returns: this if successful, a null pointer otherwise.
   333) 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).
```

ios_	base	flag co	mbinatio	n	stdio equivalent
binary	in	out	trunc	app	
		+			"W"
		+		+	"a"
				+	"a"
		+	+		"W"
	+				"r"
	+	+			"r+"
	+	+	+		"W+"
	+	+		+	"a+"
	+			+	"a+"
+		+			"wb"
+		+		+	"ab"
+				+	"ab"
+		+	+		"wb"
+	+				"rb"
+	+	+			"r+b"
+	+	+	+		"w+b"
+	+	+		+	"a+b"

+

"a+b"

Table 132 — File open modes

basic\_filebuf<charT,traits>\* open(const string& s,
 ios\_base::openmode mode);

+

Returns: open(s.c\_str(), mode);

basic\_filebuf<charT,traits>\* close();

Effects: If is\_open() == false, returns a null pointer. If a put area exists, calls overflow(traits:: eof()) to flush characters. If the last virtual member function called on \*this (between underflow, overflow, seekoff, and seekpos) was overflow then calls a\_codecvt.unshift (possibly several times) to determine a termination sequence, inserts those characters and calls overflow(traits:: eof()) again. Finally, regardless of whether any of the preceding calls fails or throws an exception, the function closes the file (as if by calling std::fclose(file)).<sup>334</sup> 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.

- 7 Returns: this on success, a null pointer otherwise.
- 8 Postcondition: is\_open() == false.

#### 27.9.1.5 Overridden virtual functions

[filebuf.virtuals]

streamsize showmanyc();

Effects: Behaves the same as basic\_streambuf::showmanyc() (27.6.3.4).

2 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.

334) The function signature fclose(FILE\*) is declared in <cstdio> (27.9.2).

```
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

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();
```

4 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).

- 6 Returns: traits::eof() to indicate failure.
- 7 Remarks: If is\_open() == false, the function always fails.
- 8 The function does not put back a character directly to the input sequence.
- 9 If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

```
int_type overflow(int_type c = traits::eof());
```

Effects: Behaves according to the description of basic\_streambuf<charT,traits>::overflow(c), except that the behavior of "consuming characters" is performed by first converting as if by:

```
charT* b = pbase();
charT* p = pptr();
charT* end;
char xbuf[XSIZE];
char* xbuf_end;
codecvt_base::result r =
   a_codecvt.out(state, b, p, end, xbuf, xbuf+XSIZE, xbuf_end);
```

and then

11

13

14

- 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 133.

Table 133 — seekoff effects

way Value	stdio Equivalent
basic_ios::beg	SEEK_SET
basic_ios::cur	SEEK_CUR
basic_ios::end	SEEK_END

Returns: A newly constructed pos\_type object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, returns pos\_type(off\_type(-1)).

```
pos_type seekpos(pos_type sp,
   ios_base::openmode which = ios_base::in | ios_base::out);
```

Alters the file position, if possible, to correspond to the position stored in sp (as described below). Altering the file position performs as follows:

- 1. if (om & ios\_base::out) != 0, then update the output sequence and write any unshift sequence;
- 2. set the file position to sp;
- 3. if (om & ios\_base::in) != 0, then update the input sequence;

where om is the open mode passed to the last call to open(). The operation fails if is\_open() returns false

- If sp is an invalid stream position, or if the function positions neither sequence, the positioning operation fails. If sp has not been obtained by a previous successful call to one of the positioning functions (seekoff or seekpos) on the same file the effects are undefined.
- Returns: sp on success. Otherwise returns pos\_type(off\_type(-1)).

```
int sync();
```

Effects: If a put area exists, calls filebuf::overflow to write the characters to the file. If a get area exists, the effect is implementation-defined.

```
void imbue(const locale& loc);
```

- Requires: If the file is not positioned at its beginning and the encoding of the current locale as determined by a\_codecvt.encoding() is state-dependent (22.4.1.4.2) then that facet is the same as the corresponding facet of loc.
- 21 Effects: Causes characters inserted or extracted after this call to be converted according to loc until another call of imbue.
- 22 Remark: This may require reconversion of previously converted characters. This in turn may require the implementation to be able to reconstruct the original contents of the file.

#### 27.9.1.6 Class template basic\_ifstream

[ifstream]

```
explicit basic_ifstream(const char* s,
           ios_base::openmode mode = ios_base::in);
       explicit basic_ifstream(const string& s,
           ios_base::openmode mode = ios_base::in);
       basic_ifstream(const basic_ifstream& rhs) = delete;
       basic_ifstream(basic_ifstream&& rhs);
       // 27.9.1.8 Assign/swap:
       basic_ifstream& operator=(const basic_ifstream& rhs) = delete;
      basic_ifstream& operator=(basic_ifstream&& rhs);
       void swap(basic_ifstream& rhs);
       // 27.9.1.9 Members:
       basic_filebuf<charT,traits>* rdbuf() const;
      bool is_open() const;
       void open(const char* s, ios_base::openmode mode = ios_base::in);
      void open(const string& s, ios_base::openmode mode = ios_base::in);
      void close();
     private:
      basic_filebuf<charT,traits> sb; // exposition only
     }:
     template <class charT, class traits>
     void swap(basic_ifstream<charT, traits>& x,
               basic_ifstream<charT, traits>& y);
   }
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]
basic_ifstream();
      Effects: Constructs an object of class basic_ifstream<charT, traits>, initializing the base class
      with basic_istream(&sb) and initializing sb with basic_filebuf<charT,traits>()) (27.7.2.1.1,
      27.9.1.2).
explicit basic_ifstream(const char* s,
     ios_base::openmode mode = ios_base::in);
      Effects: Constructs an object of class basic_ifstream, initializing the base class with basic_-
      istream(&sb) and initializing sb with basic_filebuf<charT, traits>()) (27.7.2.1.1, 27.9.1.2),
      then calls rdbuf()->open(s, mode | ios_base::in). If that function returns a null pointer, calls
      setstate(failbit).
explicit basic_ifstream(const string& s,
     ios_base::openmode mode = ios_base::in);
      Effects: the same as basic_ifstream(s.c_str(), mode).
```

1

2

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]
   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
                                                                                   [ifstream.members]
   basic_filebuf<charT,traits>* rdbuf() const;
        Returns: const_cast<basic_filebuf<charT,traits>*>(&sb).
   bool is_open() const;
        Returns: rdbuf()->is_open().
   void open(const char* s, ios_base::openmode mode = ios_base::in);
3
        Effects: Calls rdbuf()->open(s, mode | ios_base::in). If that function does not return a null
        pointer calls clear(), otherwise calls setstate(failbit) (which may throw ios_base::failure
        (27.5.5.4)).
   void open(const string& s, ios_base::openmode mode = ios_base::in);
        Effects: calls open(s.c_str(), mode).
   void close();
5
        Effects: Calls rdbuf()->close() and, if that function returns a null pointer, calls setstate(failbit)
        (which may throw ios_base::failure (27.5.5.4)).
```

# 27.9.1.10 Class template basic\_ofstream

[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(const basic_ofstream& rhs) = delete;
         basic_ofstream(basic_ofstream&& rhs);
         // 27.9.1.12  Assign/swap:
         basic_ofstream& operator=(const basic_ofstream& rhs) = delete;
         basic_ofstream& operator=(basic_ofstream&& rhs);
         void swap(basic_ofstream& rhs);
         // 27.9.1.13 Members:
         basic_filebuf<charT,traits>* rdbuf() const;
         bool is_open() const;
         void open(const char* s, ios_base::openmode mode = ios_base::out);
         void open(const string& s, ios_base::openmode mode = ios_base::out);
         void close();
       private:
         basic_filebuf<charT,traits> sb; // exposition only
       };
       template <class charT, class traits>
       void swap(basic_ofstream<charT, traits>& x,
                 basic_ofstream<charT, traits>& y);
    }
1 The class basic_ofstream<charT, traits> supports writing to named files. It uses a basic_filebuf<
  charT, traits> object to control the associated sequence. For the sake of exposition, the maintained data
  is presented here as:

 sb, the filebuf object.

  27.9.1.11 basic_ofstream constructors
                                                                                         [ofstream.cons]
  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>()) (27.7.3.2, 27.9.1.2).
  explicit basic_ofstream(const char* s,
       ios_base::openmode mode = ios_base::out);
  § 27.9.1.11
                                                                                                    1042
```

```
2
        Effects: Constructs an object of class basic_ofstream<charT, traits>, initializing the base class with
        basic_ostream(&sb) and initializing sb with basic_filebuf<charT,traits>()) (27.7.3.2, 27.9.1.2),
        then calls rdbuf()->open(s, modelios_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);
4
        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
                                                                                     [ofstream.assign]
   basic_ofstream& operator=(basic_ofstream&& 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_ofstream& rhs);
3
        Effects: Exchanges the state of *this and rhs by calling basic_ostream<charT, traits>::swap(rhs)
        and sb.swap(rhs.sb).
   template <class charT, class traits>
   void swap(basic_ofstream<charT, traits>& x,
             basic_ofstream<charT, traits>& y);
4
        Effects: x.swap(y).
   27.9.1.13 Member functions
                                                                                  [ofstream.members]
   basic_filebuf<charT,traits>* rdbuf() const;
1
        Returns: const_cast<basic_filebuf<charT,traits>*>(&sb).
   bool is_open() const;
        Returns: rdbuf()->is_open().
   void open(const char* s, ios_base::openmode mode = ios_base::out);
3
        Effects: Calls rdbuf()->open(s, mode | ios_base::out). If that function does not return a null
        pointer calls clear(), otherwise calls setstate(failbit) (which may throw ios_base::failure
        (27.5.5.4)).
```

```
void close();
4
        Effects: Calls rdbuf()->close() and, if that function fails (returns a null pointer), calls setstate(
        failbit) (which may throw ios_base::failure (27.5.5.4)).
   void open(const string& s, ios_base::openmode mode = ios_base::out);
5
        Effects: calls open(s.c_str(), mode);
   27.9.1.14 Class template basic_fstream
                                                                                               [fstream]
     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(const basic_fstream& rhs) = delete;
         basic_fstream(basic_fstream&& rhs);
         // 27.9.1.16 Assign/swap:
         basic_fstream& operator=(const basic_fstream& rhs) = delete;
         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);
     }
```

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(); Effects: Constructs an object of class basic\_fstream<charT, traits>, initializing the base class with basic\_iostream(&sb) and initializing sb with basic\_filebuf<charT,traits>(). 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\_iostream(&sb) and initializing sb with basic\_filebuf<charT, traits>(). Then calls rdbuf()->open(s, mode). If that function returns a null pointer, calls setstate(failbit). explicit basic\_fstream(const string& s, ios\_base::openmode mode = ios\_base::in|ios\_base::out); 3 Effects: the same as basic\_fstream(s.c\_str(), mode); basic\_fstream(basic\_fstream&& rhs); 4 Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic\_filebuf. Next basic\_istream<charT,traits>::set\_rdbuf(&sb) is called to install the contained basic filebuf. 27.9.1.16 Assign and swap [fstream.assign] basic\_fstream& operator=(basic\_fstream&& rhs); 1 Effects: Move assigns the base and members of \*this from the base and corresponding members of 2 Returns: \*this. void swap(basic\_fstream& rhs); 3 Effects: Exchanges the state of \*this and rhs by calling basic iostream<charT, traits>::swap(rhs) and sb.swap(rhs.sb). template <class charT, class traits> void swap(basic\_fstream<charT, traits>& x,

§ 27.9.1.16

basic\_fstream<charT, traits>& y);

Effects: x.swap(y).

#### 27.9.1.17 Member functions

[fstream.members]

```
basic_filebuf<charT,traits>* rdbuf() const;
    Returns: const_cast<basic_filebuf<charT,traits>*>(&sb).

bool is_open() const;

Returns: rdbuf()->is_open().

void open(const char* s,
    ios_base::openmode mode = ios_base::in|ios_base::out);

Effects: Calls rdbuf()->open(s,mode). If that function does not return a null pointer calls clear(),
    otherwise calls setstate(failbit), (which may throw ios_base::failure) (27.5.5.4).

void open(const string& s,
    ios_base::openmode mode = ios_base::in|ios_base::out);

Effects: calls open(s.c_str(), mode);

void close();

Effects: Calls rdbuf()->close() and, if that function returns returns a null pointer, calls setstate(
```

# 27.9.2 C library files

[c.files]

Table 134 describes header <cstdio>.

Table 134 — Header <cstdio> synopsis

failbit) (27.5.5.4) (which may throw ios\_base::failure).

Type			Name(s)		
Macros:					
BUFSIZ	FOPEN_MAX	SEEK_CUR	TMP_MAX	_IONBF	stdout
EOF	$L_{\mathtt{tmpnam}}$	SEEK_END	_IOFBF	stderr	
FILENAME_MAX	NULL <cstdio></cstdio>	SEEK_SET	_IOLBF	stdin	
Types:	FILE	fpos_t	size_t <cstdio></cstdio>		
Functions:					
clearerr	fopen	fsetpos	putc	setbuf	vprintf
fclose	fprintf	ftell	putchar	setvbuf	vscanf
feof	fputc	fwrite	puts	${ t snprintf}$	vsnprintf
ferror	fputs	getc	rename	sprintf	vsprintf
fflush	fread	getchar	remove	tmpfile	vsscanf
fgetc	freopen	gets	rewind	tmpnam	
fgetpos	fscanf	perror	scanf	ungetc	
fgets	fseek	printf	sscanf	vfprintf	

<sup>2</sup> Calls to the function tmpnam with an argument of NULL may introduce a data race (17.6.5.9) with other calls to tmpnam with an argument of NULL.

§ 27.9.2

SEE ALSO: ISO C 7.9, Amendment 1 4.6.2.

Table 135 describes header <cinttypes>. [Note: The macros defined by <cinttypes> are provided unconditionally. In particular, the symbol \_\_STDC\_FORMAT\_MACROS, mentioned in footnote 182 of the C standard, plays no role in C++. — end note]

Table 135 — Header <cinttypes> synopsis

Type	Name(s)			
Macros:				
PRI{d i	ouxX}[FAS	ST LEAST]{8	16 32 64}	
PRI{d i	o u x $X$ }{ $MAX$	X PTR}		
SCN{d i	ouxX}[FAS	ST LEAST]{8	16 32 64}	
SCN{d i	o u x $X$ }{MAX	X PTR}		
Types:	imaxdiv_t			
Function	ns:			
abs	imaxabs	${\tt strtoimax}$	wcstoimax	
div	imaxdiv	${\tt strtoumax}$	wcstoumax	

- <sup>4</sup> The contents of header <cinttypes> are the same as the Standard C Library header <inttypes.h>, with the following changes:
  - the header <cinttypes> includes the header <cstdint> instead of <stdint.h>, and
  - if and only if the type intmax\_t designates an extended integer type (3.9.1), the following function signatures are added:

```
intmax_t abs(intmax_t);
imaxdiv_t div(intmax_t, intmax_t);
```

which shall have the same semantics as the function signatures intmax\_t imaxabs(intmax\_t) and imaxdiv\_t imaxdiv(intmax\_t, intmax\_t), respectively.

§ 27.9.2

# 28 Regular expressions library

[re]

28.1 General [re.general]

<sup>1</sup> This Clause describes components that C++ programs may use to perform operations involving regular expression matching and searching.

<sup>2</sup> The following subclauses describe a basic regular expression class template and its traits that can handle char-like template arguments, two specializations of this class template that handle sequences of char and wchar\_t, a class template that holds the result of a regular expression match, a series of algorithms that allow a character sequence to be operated upon by a regular expression, and two iterator types for enumerating regular expression matches, as described in Table 136.

	Subclause	Header(s)
28.2	Definitions	
28.3	Requirements	
28.5	Constants	
28.6	Exception type	
28.7	Traits	
28.8	Regular expression template	<regex></regex>
28.9	Submatches	
28.10	Match results	
28.11	Algorithms	
28.12	Iterators	
28.13	Grammar	

28.2 Definitions [re.def]

<sup>1</sup> The following definitions shall apply to this Clause:

#### 28.2.1

[defns.regex.collating.element]

#### collating element

a sequence of one or more characters within the current locale that collate as if they were a single character.

#### 28.2.2

[defns.regex.finite.state.machine]

#### finite state machine

an unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained.

# 28.2.3

[defns.regex.format.specifier]

# format specifier

a sequence of one or more characters that is to be replaced with some part of a regular expression match.

# 28.2.4

[defns.regex.matched]

matched

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

#### [defns.regex.primary.equivalence.class]

#### primary equivalence class

a set of one or more characters which share the same primary sort key: that is the sort key weighting that depends only upon character shape, and not accents, case, or locale specific tailorings.

#### 28.2.6

[defns.regex.regular.expression]

#### regular expression

a pattern that selects specific strings from a set of character strings.

# 28.2.7

[defns.regex.subexpression]

# sub-expression

a subset of a regular expression that has been marked by parenthesis.

# 28.3 Requirements

[re.req]

- 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]
- <sup>2</sup> 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.
- <sup>3</sup> To specialize class template basic\_regex for a character container CharT and its related regular expression traits class Traits, use basic\_regex<CharT, Traits>.
- In Table 137 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 137 — Regular expression traits class requirements

Expression	Return type	Assertion/note pre-/post-condition
X::char_type charT		The character container type used in the
		implementation of class template
		basic_regex.
X::string_type	std::basic	
	string <chart></chart>	
X::locale_type	A copy	A type that represents the locale used by the
	constructible type	traits class.
X::char_class_type A bitmask A bitmask type rep		A bitmask type representing a particular
	type $(17.5.2.1.3)$ .	character classification.
X::length(p)	std::size_t	Yields the smallest $i$ such that $p[i] == 0$ .
		Complexity is linear in i.
v.translate(c)	X::char_type	Returns a character such that for any
		character d that is to be considered equivalent
		to c then v.translate(c) ==
		v.translate(d).

§ 28.3 1049

 $\odot$  ISO/IEC N3337

Table 137 — Regular expression traits class requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition
v.translate_nocase(c)	X::char_type	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).
v.transform(F1, F2)	X::string_type	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) < v.transform(H1, H2).
v.transform_primary(F1, F2)	X::string_type	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) < v.transform_primary(H1, H2).
v.lookup_collatename(F1, F2)	X::string_type	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.
v.lookup_classname(F1, F2, b)	X::char_class type	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.
v.isctype(c, cl)	bool	Returns true if character c is a member of one of the character classes designated by cl, false otherwise.
v.value(c, I)	int	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]

Table 137 — Regular expression traits class requirements (continued)

Expression Return type		Assertion/note pre-/post-condition	
u.imbue(loc)	X::locale_type	Imbues <b>u</b> with the locale <b>loc</b> and returns the	
		previous locale used by u if any.	
v.getloc()	X::locale_type	Returns the current locale used by v, if any.	

[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

[re.syn]

```
#include <initializer_list>
namespace std {
  // 28.5, regex constants:
  namespace regex_constants {
    enum error_type;
  } // namespace regex_constants
  // 28.6, class regex_error:
  class regex_error;
  // 28.7, class template regex_traits:
  template <class charT> struct regex_traits;
  // 28.8, class template basic_regex:
  template <class charT, class traits = regex_traits<charT> > class basic_regex;
 typedef basic_regex<char>
 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<</pre>
      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<</pre>
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<(</pre>
    const basic_string<</pre>
      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<</pre>
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<=(</pre>
    const basic_string<</pre>
      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<</pre>
      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<</pre>
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<(</pre>
    const sub_match<BiIter>& lhs,
    const basic_string<</pre>
      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<</pre>
      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<=(</pre>
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator <= (typename iterator_traits <BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator <= (const sub_match < BiIter > & lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator == (typename iterator_traits < BiIter >:: value_type const& lhs,
```

```
const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator == (const sub_match < BiIter > & lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class charT, class ST, class BiIter>
  basic_ostream<charT, ST>&
  operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);
// 28.10, class template match_results:
template <class BidirectionalIterator,
          class Allocator = allocator<sub_match<BidirectionalIterator> > >
  class match_results;
typedef match_results<const char*>
                                                cmatch;
typedef match_results<const wchar_t*>
                                               wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;
// match_results comparisons
template <class BidirectionalIterator, class Allocator>
  bool operator == (const match_results < BidirectionalIterator, Allocator > & m1,
                   const match_results<BidirectionalIterator, Allocator>& m2);
template <class BidirectionalIterator, class Allocator>
```

```
bool operator!= (const match_results<BidirectionalIterator, Allocator>& m1,
                   const match_results<BidirectionalIterator, Allocator>& m2);
// 28.10.7, match_results swap:
template <class BidirectionalIterator, class Allocator>
  void swap(match_results<BidirectionalIterator, Allocator>& m1,
            match_results<BidirectionalIterator, Allocator>& m2);
// 28.11.2, function template regex match:
template <class BidirectionalIterator, class Allocator,
    class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                   match_results<BidirectionalIterator, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags =
                   regex_constants::match_default);
template <class charT, class Allocator, class traits>
  bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                   match_results<
                     typename basic_string<charT, ST, SA>::const_iterator,
                     Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
template <class 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,</pre>
    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,</pre>
    class traits, class charT>
  OutputIterator
 regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits>& e,
                const charT* fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
template <class traits, class charT, class ST, class SA,
    class FST, class FSA>
  basic_string<charT, ST, SA>
  regex_replace(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits>& e,
                const basic_string<charT, FST, FSA>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
```

```
template <class traits, class charT, class ST, class SA>
      basic_string<charT, ST, SA>
      regex_replace(const basic_string<charT, ST, SA>& s,
                    const basic_regex<charT, traits>& e,
                    const charT* fmt,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
    template <class traits, class charT, class ST, class SA>
      basic_string<charT>
      regex_replace(const charT* s,
                    const basic_regex<charT, traits>& e,
                    const basic_string<charT, ST, SA>& fmt,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
    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);
    // 28.12.1, class template regex_iterator:
    template <class BidirectionalIterator,
              class charT = typename iterator_traits<</pre>
                BidirectionalIterator>::value_type,
              class traits = regex_traits<charT> >
      class regex_iterator;
    typedef regex_iterator<const char*>
                                                     cregex_iterator;
    typedef regex_iterator<const wchar_t*>
                                                     wcregex_iterator;
    typedef regex_iterator<string::const_iterator> sregex_iterator;
    typedef regex_iterator<wstring::const_iterator> wsregex_iterator;
    // 28.12.2, class template regex_token_iterator:
    template <class BidirectionalIterator,
              class charT = typename iterator_traits<</pre>
                BidirectionalIterator>::value_type,
              class traits = regex_traits<charT> >
      class regex_token_iterator;
    typedef regex_token_iterator<const char*>
                                                           cregex_token_iterator;
    typedef regex_token_iterator<const wchar_t*>
                                                           wcregex_token_iterator;
    typedef regex_token_iterator<string::const_iterator> sregex_token_iterator;
    typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;
28.5 Namespace std::regex_constants
                                                                                          [re.const]
The namespace std::regex_constants holds symbolic constants used by the regular expression library.
This namespace provides three types, syntax_option_type, match_flag_type, and error_type, along
with several constants of these types.
                                                                                         [re.synopt]
28.5.1 Bitmask type syntax_option_type
  namespace std {
    namespace regex_constants {
§ 28.5.1
                                                                                                 1057
```

```
typedef T1 syntax_option_type;
static constexpr syntax_option_type icase = unspecified;
static constexpr syntax_option_type nosubs = unspecified;
static constexpr syntax_option_type optimize = unspecified;
static constexpr syntax_option_type collate = unspecified;
static constexpr syntax_option_type ECMAScript = unspecified;
static constexpr syntax_option_type basic = unspecified;
static constexpr syntax_option_type extended = unspecified;
static constexpr syntax_option_type awk = unspecified;
static constexpr syntax_option_type grep = unspecified;
static constexpr syntax_option_type egrep = unspecified;
static constexpr syntax_option_type egrep = unspecified;
}
```

The type syntax\_option\_type is an implementation-defined bitmask type (17.5.2.1.3). Setting its elements has the effects listed in table 138. A valid value of type syntax\_option\_type shall have exactly one of the elements ECMAScript, basic, extended, awk, grep, egrep, set.

Table 138 —	syntax	option	type	effects
-------------	--------	--------	------	---------

Element	Effect(s) if set
icase	Specifies that matching of regular expressions against a character container
	sequence shall be performed without regard to case.
nosubs	Specifies that when a regular expression is matched against a character con-
	tainer sequence, no sub-expression matches shall be stored in the supplied
	match_results structure.
optimize	Specifies that the regular expression engine should pay more attention to
	the speed with which regular expressions are matched, and less to the speed
	with which regular expression objects are constructed. Otherwise it has no
	detectable effect on the program output.
collate	Specifies that character ranges of the form "[a-b]" shall be locale sensitive.
ECMAScript	Specifies that the grammar recognized by the regular expression engine
	shall be that used by ECMAScript in ECMA-262, as modified in 28.13.
basic	Specifies that the grammar recognized by the regular expression engine
	shall be that used by basic regular expressions in POSIX, Base Definitions
	and Headers, Section 9, Regular Expressions.
extended	Specifies that the grammar recognized by the regular expression engine shall
	be that used by extended regular expressions in POSIX, Base Definitions
	and Headers, Section 9, Regular Expressions.
awk	Specifies that the grammar recognized by the regular expression engine
	shall be that used by the utility awk in POSIX.
grep	Specifies that the grammar recognized by the regular expression engine
	shall be that used by the utility grep in POSIX.
egrep	Specifies that the grammar recognized by the regular expression engine
	shall be that used by the utility grep when given the -E option in POSIX.

# 28.5.2 Bitmask type regex\_constants::match\_flag\_type

[re.matchflag]

namespace std {

§ 28.5.2 1058

```
namespace regex_constants{
    typedef T2 match_flag_type;
    static constexpr match_flag_type match_default = 0
    static constexpr match_flag_type match_not_bol = unspecified ;
    static constexpr match_flag_type match_not_eol = unspecified ;
    static constexpr match_flag_type match_not_bow = unspecified ;
    static constexpr match_flag_type match_not_eow = unspecified ;
    static constexpr match_flag_type match_any = unspecified ;
    static constexpr match_flag_type match_not_null = unspecified ;
    static constexpr match_flag_type match_continuous = unspecified ;
    static constexpr match_flag_type match_prev_avail = unspecified ;
    static constexpr match_flag_type format_default = 0;
    static constexpr match_flag_type format_sed = unspecified ;
    static constexpr match_flag_type format_no_copy = unspecified ;
    static constexpr match_flag_type format_first_only = unspecified ;
}
```

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 139 for any bitmask elements set.

Table 139 — regex\_constants::match\_flag\_type effects when obtaining a match against a character container sequence [first, last).

Element	Effect(s) if set
match_not_bol	The first character in the sequence [first,last) shall be treated as though
	it is not at the beginning of a line, so the character ^ in the regular expres-
	sion shall not match [first,first).
match_not_eol	The last character in the sequence [first,last) shall be treated as though
	it is not at the end of a line, so the character "\$" in the regular expression
	shall not match [last,last).
match_not_bow	The expression "\b" shall not match the sub-sequence [first,first).
match_not_eow	The expression "\b" shall not match the sub-sequence [last,last).
match_any	If more than one match is possible then any match is an acceptable result.
match_not_null	The expression shall not match an empty sequence.
match_continuous	The expression shall only match a sub-sequence that begins at first.
match_prev_avail	first is a valid iterator position. When this flag is set the flags match
	not_bol and match_not_bow shall be ignored by the regular expression
	algorithms 28.11 and iterators 28.12.
format_default	When a regular expression match is to be replaced by a new string, the
	new string shall be constructed using the rules used by the ECMAScript
	replace function in ECMA-262, part 15.5.4.11 String.prototype.replace. In
	addition, during search and replace operations all non-overlapping occur-
	rences 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.
format_sed	When a regular expression match is to be replaced by a new string, the
	new string shall be constructed using the rules used by the sed utility in
	POSIX.

§ 28.5.2

Table 139 — regex\_constants::match\_flag\_type effects when obtaining a match against a character container sequence [first, last). (continued)

Element	$\operatorname{Effect}(s)$ if set
format_no_copy	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.
format_first_only	When specified during a search and replace operation, only the first occurrence of the regular expression shall be replaced.

# 28.5.3 Implementation-defined error\_type

[re.err]

```
namespace std {
  namespace regex_constants {
    typedef T3 error_type;
    static constexpr error_type error_collate = unspecified ;
    static constexpr error_type error_ctype = unspecified ;
    static constexpr error_type error_escape = unspecified ;
    static constexpr error_type error_backref = unspecified ;
    static constexpr error_type error_brack = unspecified ;
    static constexpr error_type error_paren = unspecified ;
    static constexpr error_type error_brace = unspecified ;
    static constexpr error_type error_badbrace = unspecified ;
    static constexpr error_type error_range = unspecified ;
    static constexpr error_type error_space = unspecified ;
    static constexpr error_type error_badrepeat = unspecified ;
    static constexpr error_type error_complexity = unspecified ;
    static constexpr error_type error_stack = unspecified ;
}
```

The type error\_type is an implementation-defined enumerated type (17.5.2.1.2). Values of type error\_type represent the error conditions described in Table 140:

Table 140 —	error	type	values	in	the	$\mathbf{C}$	$local\epsilon$	•

Value	Error condition
error_collate	The expression contained an invalid collating element name.
error_ctype	The expression contained an invalid character class name.
error_escape	The expression contained an invalid escaped character, or a trailing escape.
error_backref	The expression contained an invalid back reference.
error_brack	The expression contained mismatched [ and ].
error_paren	The expression contained mismatched ( and ).
error_brace	The expression contained mismatched { and }
error_badbrace	The expression contained an invalid range in a {} expression.
error_range	The expression contained an invalid character range, such as [b-a] in most
	encodings.
error_space	There was insufficient memory to convert the expression into a finite state
	machine.
error_badrepeat	One of *?+{ was not preceded by a valid regular expression.

§ 28.5.3

Table 140 — error\_type values in the C locale (continued)

Value	Error condition
error_complexity	The complexity of an attempted match against a regular expression ex-
	ceeded a pre-set level.
error_stack	There was insufficient memory to determine whether the regular expression
	could match the specified character sequence.

## 28.6 Class regex\_error

2

3

[re.badexp]

```
class regex_error : public std::runtime_error {
  public:
    explicit regex_error(regex_constants::error_type ecode);
    regex_constants::error_type code() const;
};
```

The class regex\_error defines the type of objects thrown as exceptions to report errors from the regular expression library.

```
regex_error(regex_constants::error_type ecode);
    Effects: Constructs an object of class regex_error.
    Postcondition:: ecode == code()
```

*Returns:* The error code that was passed to the constructor.

# 28.7 Class template regex\_traits

regex\_constants::error\_type code() const;

[re.traits]

```
namespace std {
  template <class charT>
 struct regex_traits {
  public:
     typedef charT
                                          char_type;
     typedef std::basic_string<char_type> string_type;
     typedef std::locale
                                          locale_type;
     typedef bitmask_type
                                          char_class_type;
     regex_traits();
     static std::size_t length(const char_type* p);
     charT translate(charT c) const;
     charT translate_nocase(charT c) const;
     template <class ForwardIterator>
       string_type transform(ForwardIterator first, ForwardIterator last) const;
     template <class ForwardIterator>
       string_type transform_primary(
         ForwardIterator first, ForwardIterator last) const;
     template <class ForwardIterator>
       string_type lookup_collatename(
         ForwardIterator first, ForwardIterator last) const;
     template <class ForwardIterator>
       char_class_type lookup_classname(
         ForwardIterator first, ForwardIterator last, bool icase = false) const;
     bool isctype(charT c, char_class_type f) const;
```

§ 28.7 1061

int value(charT ch, int radix) const;

```
locale_type imbue(locale_type 1);
          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);
3
        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;
6
        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;
8
        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;
   § 28.7
                                                                                                    1062
```

Returns: an unspecified value that represents the character classification named by the character sequence designated by the iterator range [first,last). If the parameter icase is true then the returned mask identifies the character classification without regard to the case of the characters being matched, otherwise it does honor the case of the characters being matched. The value returned shall be independent of the case of the characters in the character sequence. If the name is not recognized then returns a value that compares equal to 0.

Remarks: For regex\_traits<char>, at least the names "d", "w", "s", "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit" shall be recognized. For regex\_traits<wchar\_t>, at least the names L"d", L"w", L"s", L"alnum", L"alpha", L"blank", L"cntrl", L"digit", L"graph", L"lower", L"print", L"punct", L"space", L"upper" and L"xdigit" shall be recognized.

bool isctype(charT c, char\_class\_type f) const;

11 Effects: Determines if the character c is a member of the character classification represented by f.

Returns: Converts f into a value m of type std::ctype\_base::mask in an unspecified manner, and returns true if use\_facet<ctype<charT> >(getloc()).is(m, c) is true. Otherwise returns true if f bitwise or'ed with the result of calling lookup\_classname with an iterator pair that designates the character sequence "w" is not equal to 0 and c == '\_', or if f bitwise or'ed with the result of calling lookup\_classname with an iterator pair that designates the character sequence "blank" is not equal to 0 and c is one of an implementation-defined subset of the characters for which isspace(c, getloc()) returns true, otherwise returns false.

int value(charT ch, int radix) const;

13 Requires: The value of radix shall be 8, 10, or 16.

Returns: the value represented by the digit ch in base radix if the character ch is a valid digit in base radix; otherwise returns -1.

locale\_type imbue(locale\_type loc);

Effects: Imbues this with a copy of the locale loc. [Note: Calling imbue with a different locale than the one currently in use invalidates all cached data held by \*this. — end note]

Returns: if no locale has been previously imbued then a copy of the global locale in effect at the time of construction of \*this, otherwise a copy of the last argument passed to imbue.

Postcondition: getloc() == loc.

locale\_type getloc()const;

18

Returns: if no locale has been imbued then a copy of the global locale in effect at the time of construction of \*this, otherwise a copy of the last argument passed to imbue.

335) For example, if the parameter icase is true then [[:lower:]] is the same as [[:alpha:]].

§ 28.7 1063

## 28.8 Class template basic\_regex

[re.regex]

For a char-like type charT, specializations of class template basic\_regex represent regular expressions constructed from character sequences of charT characters. In the rest of 28.8, charT denotes a given charlike type. Storage for a regular expression is allocated and freed as necessary by the member functions of class basic\_regex.

- Objects of type specialization of basic\_regex are responsible for converting the sequence of charT objects to an internal representation. It is not specified what form this representation takes, nor how it is accessed by algorithms that operate on regular expressions. [Note: Implementations will typically declare some function templates as friends of basic regex to achieve this end note]
- 3 The functions described in this Clause report errors by throwing exceptions of type regex error.

```
namespace std {
 template <class charT,
            class traits = regex_traits<charT> >
 class basic_regex {
 public:
   // types:
   typedef
                     charT
                                                           value_type;
    typedef
                     traits
                                                           traits_type;
    typedef typename traits::string_type
                                                           string_type;
                    regex_constants::syntax_option_type flag_type;
    typedef
    typedef typename traits::locale_type
                                                           locale_type;
    // 28.8.1, constants:
    static constexpr regex_constants::syntax_option_type
      icase = regex_constants::icase;
    static constexpr regex_constants::syntax_option_type
     nosubs = regex_constants::nosubs;
    static constexpr regex_constants::syntax_option_type
      optimize = regex_constants::optimize;
    static constexpr regex_constants::syntax_option_type
      collate = regex_constants::collate;
    static constexpr regex_constants::syntax_option_type
      ECMAScript = regex_constants::ECMAScript;
    static constexpr regex_constants::syntax_option_type
      basic = regex_constants::basic;
    static constexpr regex_constants::syntax_option_type
      extended = regex_constants::extended;
    static constexpr regex_constants::syntax_option_type
      awk = regex_constants::awk;
    static constexpr regex_constants::syntax_option_type
      grep = regex_constants::grep;
    static constexpr regex_constants::syntax_option_type
      egrep = regex_constants::egrep;
    // 28.8.2, construct/copy/destroy:
    basic_regex();
    explicit basic_regex(const charT* p,
      flag_type f = regex_constants::ECMAScript);
    basic_regex(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);
    basic_regex(const basic_regex&);
    basic_regex(basic_regex&&) noexcept;
    template <class ST, class SA>
      explicit basic_regex(const basic_string<charT, ST, SA>& p,
                           flag_type f = regex_constants::ECMAScript);
```

§ 28.8 1064

template <class ForwardIterator>

```
basic_regex(ForwardIterator first, ForwardIterator last,
                    flag_type f = regex_constants::ECMAScript);
      basic_regex(initializer_list<charT>,
        flag_type = regex_constants::ECMAScript);
      ~basic_regex();
      basic_regex& operator=(const basic_regex&);
      basic_regex& operator=(basic_regex&&) noexcept;
      basic_regex& operator=(const charT* ptr);
      basic_regex& operator=(initializer_list<charT> il);
      template <class ST, class SA>
        basic_regex& operator=(const basic_string<charT, ST, SA>& p);
      // 28.8.3, assign:
      basic_regex& assign(const basic_regex& that);
      basic_regex& assign(basic_regex&& that) noexcept;
      basic_regex& assign(const charT* ptr,
        flag_type f = regex_constants::ECMAScript);
      basic_regex& assign(const charT* p, size_t len, flag_type f);
      template <class string_traits, class A>
        basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                            flag_type f = regex_constants::ECMAScript);
      template <class InputIterator>
        basic_regex& assign(InputIterator first, InputIterator last,
                            flag_type f = regex_constants::ECMAScript);
      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
                                                                                   [re.regex.const]
  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
§ 28.8.1
                                                                                                 1065
```

```
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;
<sup>1</sup> The static constant members are provided as synonyms for the constants declared in namespace regex -
   constants.
   28.8.2
                                                                                    [re.regex.construct]
           basic regex constructors
   basic_regex();
        Effects: Constructs an object of class basic regex that does not match any character sequence.
   basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
         Requires: p shall not be a null pointer.
         Throws: regex_error if p is not a valid regular expression.
         Effects: Constructs an object of class basic_regex; the object's internal finite state machine is con-
        structed from the regular expression contained in the array of charT of length char_traits<charT>::
        length(p) whose first element is designated by p, and interpreted according to the flags f.
        Postconditions: flags() returns f. mark_count() returns the number of marked sub-expressions
        within the expression.
   basic_regex(const charT* p, size_t len, flag_type f);
         Requires: p shall not be a null pointer.
         Throws: regex_error if p is not a valid regular expression.
        Effects: Constructs an object of class basic_regex; the object's internal finite state machine is con-
        structed from the regular expression contained in the sequence of characters [p,p+len), and interpreted
        according the flags specified in f.
        Postconditions: flags() returns f. mark_count() returns the number of marked sub-expressions
        within the expression.
   basic_regex(const basic_regex& e);
        Effects: Constructs an object of class basic_regex as a copy of the object e.
        Postconditions: flags() and mark_count() return e.flags() and e.mark_count(), respectively.
   basic_regex(basic_regex&& e) noexcept;
         Effects: Move constructs an object of class basic_regex from e.
```

1

2

3

4

5

6

7

8

9

10

11

12

13

§ 28.8.2 1066

respectively, had before construction. e is in a valid state with unspecified value.

Postconditions: flags() and mark\_count() return the values that e.flags() and e.mark\_count(),

```
template <class ST, class SA>
     basic_regex(const basic_string<charT, ST, SA>& s,
                  flag_type f = regex_constants::ECMAScript);
14
         Throws: regex_error if s is not a valid regular expression.
15
         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 string s, and interpreted according to the flags
         specified in f.
16
         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);
17
         Throws: regex_error if the sequence [first,last) is not a valid regular expression.
18
         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.
19
         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);
20
         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) noexcept;
2
         Effects: returns assign(std::move(e)).
   basic_regex& operator=(const charT* ptr);
3
         Requires: ptr shall not be a null pointer.
         Effects: returns assign(ptr).
4
   basic_regex& operator=(initializer_list<charT> il);
5
         Effects: returns assign(il.begin(), il.end()).
   template <class ST, class SA>
     basic_regex& operator=(const basic_string<charT, ST, SA>& p);
   § 28.8.3
                                                                                                      1067
```

```
6
         Effects: returns assign(p).
   basic_regex& assign(const basic_regex& that);
7
         Effects: copies that into *this and returns *this.
8
         Postconditions: flags() and mark_count() return that.flags() and that.mark_count(), respec-
         tively.
   basic_regex& assign(basic_regex&& that) noexcept;
9
         Effects: move assigns from that into *this and returns *this.
10
         Postconditions: flags() and mark count() return the values that that.flags() and that.mark -
         count(), respectively, had before assignment. that is in a valid state with unspecified value.
   basic_regex& assign(const charT* ptr, flag_type f = regex_constants::ECMAScript);
11
         Returns: assign(string_type(ptr), f).
   basic_regex& assign(const charT* ptr, size_t len,
      flag_type f = regex_constants::ECMAScript);
12
         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);
13
         Throws: regex_error if s is not a valid regular expression.
14
         Returns: *this.
15
         Effects: Assigns the regular expression contained in the string s, interpreted according the flags spec-
         ified in f. If an exception is thrown, *this is unchanged.
16
         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);
17
         Requires: The type InputIterator shall satisfy the requirements for an Input Iterator (24.2.3).
18
         Returns: assign(string_type(first, last), f).
   basic_regex& assign(initializer_list<charT> il,
                        flag_type f = regex_constants::ECMAScript);
19
         Effects: Same as assign(il.begin(), il.end(), f).
20
         Returns: *this.
```

§ 28.8.3

```
28.8.4 basic_regex constant operations
```

[re.regex.operations]

unsigned mark\_count() const;

Effects: Returns the number of marked sub-expressions within the regular expression.

flag\_type flags() const;

1

1

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

[re.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

[re.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.
- 3 Complexity: constant time.

## 28.8.7 basic\_regex non-member functions

[re.regex.nonmemb]

## 28.8.7.1 basic\_regex non-member swap

[re.regex.nmswap]

```
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

[re.submatch]

Class template sub\_match denotes the sequence of characters matched by a particular marked sub-expression.

```
bool matched;
         constexpr sub_match();
         difference_type length() const;
         operator string_type() const;
         string_type str() const;
         int compare(const sub_match& s) const;
         int compare(const string_type& s) const;
         int compare(const value_type* s) const;
      };
    }
   28.9.1 sub_match members
                                                                          [re.submatch.members]
   constexpr sub_match();
        Effects: Value-initializes the pair base class subobject and the member matched.
   difference_type length() const;
2
        Returns: (matched ? distance(first, second) : 0).
   operator string_type() const;
3
        Returns: matched ? string_type(first, second) : string_type().
   string_type str() const;
4
        Returns: matched ? string_type(first, second) : string_type().
   int compare(const sub_match& s) const;
5
        Returns: str().compare(s.str()).
   int compare(const string_type& s) const;
        Returns: str().compare(s).
   int compare(const value_type* s) const;
7
        Returns: str().compare(s).
```

## 28.9.2 sub\_match non-member operators

[re.submatch.op]

```
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>& 1hs, 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<</pre>
         typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
       const sub_match<BiIter>& rhs);
        Returns: rhs.compare(lhs.c_str()) == 0.
   template <class BiIter, class ST, class SA>
     bool operator!=(
       const basic_string<</pre>
         typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
       const sub_match<BiIter>& rhs);
8
        Returns: !(lhs == rhs).
```

```
template <class BiIter, class ST, class SA>
     bool operator<(
        const basic_string<</pre>
          typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
        const sub_match<BiIter>& rhs);
9
         Returns: rhs.compare(lhs.c_str()) > 0.
   template <class BiIter, class ST, class SA>
     bool operator>(
        const basic_string<</pre>
          typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
        const sub_match<BiIter>& rhs);
10
         Returns: rhs < lhs.
   template <class BiIter, class ST, class SA>
     bool operator>=(
        const basic_string<</pre>
          typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
        const sub_match<BiIter>& rhs);
11
         Returns: !(lhs < rhs).
   template <class BiIter, class ST, class SA>
     bool operator<=(</pre>
       const basic_string<</pre>
          typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
        const sub_match<BiIter>& rhs);
         Returns: !(rhs < lhs).
12
   template <class BiIter, class ST, class SA>
     bool operator==(const sub_match<BiIter>& lhs,
                      const basic_string<</pre>
                        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
13
         Returns: lhs.compare(rhs.c_str()) == 0.
   template <class BiIter, class ST, class SA>
     bool operator!=(const sub_match<BiIter>& lhs,
                      const basic_string<
                        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
14
         Returns: !(lhs == rhs).
   template <class BiIter, class ST, class SA>
     bool operator<(const sub_match<BiIter>& lhs,
                     const basic_string<</pre>
                       typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

```
15
         Returns: lhs.compare(rhs.c_str()) < 0.
   template <class BiIter, class ST, class SA>
      bool operator>(const sub_match<BiIter>& lhs,
                     const basic_string<</pre>
                       typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
16
         Returns: rhs < lhs.
   template <class BiIter, class ST, class SA>
      bool operator>=(const sub_match<BiIter>& lhs,
                      const basic_string<</pre>
                        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
17
         Returns: !(lhs < rhs).
   template <class BiIter, class ST, class SA>
     bool operator <= (const sub_match < BiIter > & lhs,
                      const basic_string<</pre>
                        typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
18
         Returns: !(rhs < lhs).
   template <class BiIter>
     bool operator == (typename iterator_traits < BiIter >:: value_type const * lhs,
                      const sub_match<BiIter>& rhs);
19
         Returns: rhs.compare(lhs) == 0.
   template <class BiIter>
     bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
                      const sub_match<BiIter>& rhs);
20
         Returns: !(lhs == rhs).
   template <class BiIter>
     bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
                     const sub_match<BiIter>& rhs);
21
         Returns: rhs.compare(lhs) > 0.
   template <class BiIter>
     bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
                     const sub_match<BiIter>& rhs);
22
         Returns: rhs < lhs.
   template <class BiIter>
     bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
                      const sub_match<BiIter>& rhs);
```

```
23
         Returns: !(lhs < rhs).
   template <class BiIter>
     bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
                      const sub_match<BiIter>& rhs);
24
         Returns: !(rhs < lhs).
   template <class BiIter>
     bool operator==(const sub_match<BiIter>& lhs,
                      typename iterator_traits<BiIter>::value_type const* rhs);
25
         Returns: lhs.compare(rhs) == 0.
   template <class BiIter>
     bool operator!=(const sub_match<BiIter>& lhs,
                      typename iterator_traits<BiIter>::value_type const* rhs);
26
         Returns: !(lhs == rhs).
   template <class BiIter>
     bool operator<(const sub_match<BiIter>& lhs,
                     typename iterator_traits<BiIter>::value_type const* rhs);
27
         Returns: lhs.compare(rhs) < 0.
   template <class BiIter>
     bool operator>(const sub_match<BiIter>& lhs,
                     typename iterator_traits<BiIter>::value_type const* rhs);
28
         Returns: rhs < lhs.
   template <class BiIter>
     bool operator>=(const sub_match<BiIter>& lhs,
                      typename iterator_traits<BiIter>::value_type const* rhs);
29
         Returns: !(lhs < rhs).
   template <class BiIter>
     bool operator<=(const sub_match<BiIter>& lhs,
                      typename iterator_traits<BiIter>::value_type const* rhs);
         Returns: !(rhs < lhs).
   template <class BiIter>
     bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
                      const sub_match<BiIter>& rhs);
         Returns: rhs.compare(typename sub_match<BiIter>::string_type(1, lhs)) == 0.
```

```
template <class BiIter>
     bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
                      const sub_match<BiIter>& rhs);
32
         Returns: !(lhs == rhs).
   template <class BiIter>
     bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
                    const sub_match<BiIter>& rhs);
33
         Returns: rhs.compare(typename sub_match<BiIter>::string_type(1, lhs)) > 0.
   template <class BiIter>
     bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
                    const sub_match<BiIter>& rhs);
34
         Returns: rhs < lhs.
   template <class BiIter>
     bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
                      const sub_match<BiIter>& rhs);
35
         Returns: !(lhs < rhs).
   template <class BiIter>
     bool operator <= (typename iterator_traits < BiIter >:: value_type const& lhs,
                      const sub_match<BiIter>& rhs);
36
         Returns: !(rhs < lhs).
   template <class BiIter>
     bool operator==(const sub_match<BiIter>& lhs,
                      typename iterator_traits<BiIter>::value_type const& rhs);
37
         Returns: lhs.compare(typename sub_match<BiIter>::string_type(1, rhs)) == 0.
   template <class BiIter>
     bool operator!=(const sub_match<BiIter>& lhs,
                      typename iterator_traits<BiIter>::value_type const& rhs);
38
         Returns: !(lhs == rhs).
   template <class BiIter>
     bool operator<(const sub_match<BiIter>& lhs,
                     typename iterator_traits<BiIter>::value_type const& rhs);
39
         Returns: lhs.compare(typename sub_match<BiIter>::string_type(1, rhs)) < 0.
```

```
template <class BiIter>
     bool operator>(const sub_match<BiIter>& lhs,
                     typename iterator_traits<BiIter>::value_type const& rhs);
40
         Returns: rhs < lhs.
   template <class BiIter>
     bool operator>=(const sub_match<BiIter>& lhs,
                      typename iterator_traits<BiIter>::value_type const& rhs);
41
         Returns: !(lhs < rhs).
   template <class BiIter>
     bool operator<=(const sub_match<BiIter>& lhs,
                      typename iterator_traits<BiIter>::value_type const& rhs);
42
         Returns: !(rhs < lhs).
   template <class charT, class ST, class BiIter>
     basic_ostream<charT, ST>&
      operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);
43
         Returns: (os << m.str()).
```

## 28.10 Class template match\_results

[re.results]

- Class template match\_results denotes a collection of character sequences representing the result of a regular expression match. Storage for the collection is allocated and freed as necessary by the member functions of class template match\_results.
- <sup>2</sup> 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.
- <sup>3</sup> A default-constructed match\_results object has no fully established result state. A match result is ready when, as a consequence of a completed regular expression match modifying such an object, its result state becomes fully established. The effects of calling most member functions from a match\_results object that is not ready are undefined.
- The sub\_match object stored at index 0 represents sub-expression 0, i.e., the whole match. In this case the sub\_match member matched is always true. The sub\_match object stored at index n denotes what matched the marked sub-expression n within the matched expression. If the sub-expression n participated in a regular expression match then the sub\_match member matched evaluates to true, and members first and second denote the range of characters [first,second) which formed that match. Otherwise matched is false, and members first and second point to the end of the sequence that was searched. [Note: The sub\_match objects representing different sub-expressions that did not participate in a regular expression match need not be distinct.—end note]

§ 28.10

```
typedef implementation-defined
                                                                 const iterator;
typedef const_iterator
                                                                 iterator:
typedef typename
iterator_traits<BidirectionalIterator>::difference_type
                                                                 difference_type;
typedef typename allocator_traits<Allocator>::size_type
                                                                 size_type;
typedef Allocator
                                                                 allocator_type;
typedef typename iterator_traits<BidirectionalIterator>::
 value type
                                                                 char_type;
typedef basic_string<char_type>
                                                                 string_type;
// 28.10.1, construct/copy/destroy:
explicit match_results(const Allocator& a = Allocator());
match_results(const match_results& m);
match_results(match_results&& m) noexcept;
match_results& operator=(const match_results& m);
match_results& operator=(match_results&& m);
~match_results();
// 28.10.2, state:
bool ready() const;
// 28.10.3, size:
size_type size() const;
size_type max_size() const;
bool empty() const;
// 28.10.4, element access:
difference_type length(size_type sub = 0) const;
difference_type position(size_type sub = 0) const;
string_type str(size_type sub = 0) const;
const_reference operator[](size_type n) const;
const_reference prefix() const;
const_reference suffix() const;
const_iterator begin() const;
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;
// 28.10.5, format:
template <class OutputIter>
OutputIter
format(OutputIter out,
        const char_type* fmt_first, const char_type* fmt_last,
        regex_constants::match_flag_type flags =
         regex_constants::format_default) const;
template <class OutputIter, class ST, class SA>
 OutputIter
 format(OutputIter out,
         const basic_string<char_type, ST, SA>& fmt,
         regex_constants::match_flag_type flags =
           regex_constants::format_default) const;
template <class ST, class SA>
basic_string<char_type, ST, SA>
format(const basic_string<char_type, ST, SA>& fmt,
```

§ 28.10 1077

```
regex_constants::match_flag_type flags =
                    regex_constants::format_default) const;
          string_type
          format(const char_type* fmt,
                 regex_constants::match_flag_type flags =
                   regex_constants::format_default) const;
          // 28.10.6, allocator:
          allocator_type get_allocator() const;
          // 28.10.7, swap:
          void swap(match_results& that);
       };
     }
             match_results constructors
                                                                                      [re.results.const]
   28.10.1
  In all match_results constructors, a copy of the Allocator argument shall be used for any memory allo-
   cation 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.
3
        Postconditions: ready() returns false. size() returns 0.
   match_results(const match_results& m);
        Effects: Constructs an object of class match_results, as a copy of m.
   match_results(match_results&& m) noexcept;
5
        Effects: Move-constructs an object of class match_results from m satisfying the same postconditions
        as Table 141. Additionally, the stored Allocator value is move constructed from m.get_allocator().
6
        Throws: Nothing if the allocator's move constructor throws nothing.
   match_results& operator=(const match_results& m);
7
        Effects: Assigns m to *this. The postconditions of this function are indicated in Table 141.
   match_results& operator=(match_results&& m);
        Effects: Move-assigns m to *this. The postconditions of this function are indicated in Table 141.
   28.10.2 match_results state
                                                                                       [re.results.state]
   bool ready() const;
        Returns: true if *this has a fully established result state, otherwise false.
```

§ 28.10.2

(c) ISO/IEC N3337

Element	Value	
ready()	m.ready()	
size()	m.size()	
str(n)	m.str(n) for all integers n < m.size()	
<pre>prefix()</pre>	m.prefix()	
suffix()	m.suffix()	
(*this)[n]	m[n] for all integers n < m.size()	
length(n)	m.length(n) for all integers n < m.size()	
position(n)	<pre>m.position(n) for all integers n &lt; m.size()</pre>	

Table 141 — match\_results assignment operator effects

## 28.10.3 match\_results size

[re.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;
```

2 Returns: The maximum number of sub\_match elements that can be stored in \*this.

```
bool empty() const;
Returns: size() == 0.
```

§ 28.10.4

## 28.10.4 match\_results element access

[re.results.acc]

1079

```
difference_type length(size_type sub = 0) const;
    Requires: ready() == true.
    Returns: (*this)[sub].length().

difference_type position(size_type sub = 0) const;

Requires: ready() == true.

Returns: The distance from the start of the target sequence to (*this)[sub].first.

string_type str(size_type sub = 0) const;

Requires: ready() == true.

Returns: string_type((*this)[sub]).

const_reference operator[](size_type n) const;
```

```
7 Requires: ready() == true.
```

Returns: A reference to the sub\_match object representing the character sequence that matched marked sub-expression n. If n == 0 then returns a reference to a sub\_match object representing the character sequence that matched the whole regular expression. If n >= size() then returns a sub\_match object representing an unmatched sub-expression.

```
const_reference prefix() const;
```

- 9 Requires: ready() == true.
- Returns: A reference to the sub\_match object representing the character sequence from the start of the string being matched/searched to the start of the match found.

```
const_reference suffix() const;
```

- 11 Requires: ready() == true.
- Returns: A reference to the sub\_match object representing the character sequence from the end of the match found to the end of the string being matched/searched.

```
const_iterator begin() const;
const_iterator cbegin() const;
```

Returns: A starting iterator that enumerates over all the sub-expressions stored in \*this.

```
const_iterator end() const;
const_iterator cend() const;
```

13

14 Returns: A terminating iterator that enumerates over all the sub-expressions stored in \*this.

## 28.10.5 match\_results formatting

[re.results.form]

- Requires: ready() == true and OutputIter shall satisfy the requirements for an Output Iterator (24.2.4).
- 2 Effects: Copies the character sequence [fmt\_first,fmt\_last) to OutputIter out. Replaces each format specifier or escape sequence in the copied range with either the character(s) it represents or the sequence of characters within \*this to which it refers. The bitmasks specified in flags determine which format specifiers and escape sequences are recognized.
- 3 Returns: out.

§ 28.10.5

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;
5
         Requires: ready() == true.
6
         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;
8
         Requires: ready() == true.
9
         Effects:
                  Constructs
                              an
                                     empty
                                              string
                                                      result of type string_type
        format(back_inserter(result), fmt, fmt + char_traits<char_type>::length(fmt), flags).
10
         Returns: result.
   28.10.6
              match_results allocator
                                                                                        [re.results.all]
   allocator_type get_allocator() const;
1
         Returns: A copy of the Allocator that was passed to the object's constructor or, if that allocator has
        been replaced, a copy of the most recent replacement.
   28.10.7
             match_results swap
                                                                                     [re.results.swap]
   void swap(match_results& that);
1
         Effects: Swaps the contents of the two sequences.
2
         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.
3
         Complexity: constant time.
   template <class BidirectionalIterator, class Allocator>
     void swap(match_results<BidirectionalIterator, Allocator>& m1,
               match_results<BidirectionalIterator, Allocator>& m2);
  Effects: m1.swap(m2).
              match_results non-member functions
                                                                             [re.results.nonmember]
   28.10.8
   template <class BidirectionalIterator, class Allocator>
   bool operator == (const match_results < Bidirectional Iterator, Allocator > & m1,
                   const match_results<BidirectionalIterator, Allocator>& m2);
1
         Returns: true if neither match result is ready, false if one match result is ready and the other is not.
        If both match results are ready, returns true only if:
          — m1.empty() && m2.empty(), or
```

§ 28.10.8

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

1

[re.alg.match]

- Requires: The type BidirectionalIterator shall satisfy the requirements of a Bidirectional Iterator (24.2.6).
- 2 Effects: Determines whether there is a match between the regular expression e, and all of the character sequence [first,last). The parameter flags is used to control how the expression is matched against the character sequence. Returns true if such a match exists, false otherwise.
- Postconditions: m.ready() == true in all cases. If the function returns false, then the effect on parameter m is unspecified except that m.size() returns 0 and m.empty() returns true. Otherwise the effects on parameter m are given in Table 142.

Element	Value
m.size()	1 + e.mark_count()
m.empty()	false
m.prefix().first	first
m.prefix().second	first
m.prefix().matched	false
m.suffix().first	last
m.suffix().second	last
m.suffix().matched	false
m[0].first	first
m[0].second	last

Table 142 — Effects of regex\_match algorithm

Element	Value	
m[0].matched	true if a full match was found.	
m[n].first	For all integers n < m.size(), the start of the sequence	
	that matched sub-expression n. Alternatively, if sub-	
	expression n did not participate in the match, then last.	
m[n].second	For all integers n < m.size(), the end of the sequence that	
	matched sub-expression n. Alternatively, if sub-expression	
	n did not participate in the match, then last.	
m[n].matched	For all integers n < m.size(), true if sub-expression n	
	participated in the match, false otherwise.	

Table 142 — Effects of regex\_match algorithm (continued)

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).

5 Returns: regex\_match(str, str + char\_traits<charT>::length(str), m, e, flags).

Returns: regex\_match(s.begin(), s.end(), m, e, flags).

7 Returns: regex\_match(str, str + char\_traits<charT>::length(str), e, flags)

Returns: regex\_match(s.begin(), s.end(), e, flags).

## 28.11.3 regex\_search

8

[re.alg.search]

- Requires: Type BidirectionalIterator shall satisfy the requirements of a Bidirectional Iterator (24.2.6).
- 2 Effects: Determines whether there is some sub-sequence within [first,last) that matches the regular expression e. The parameter flags is used to control how the expression is matched against the character sequence. Returns true if such a sequence exists, false otherwise.
- Postconditions: m.ready() == true in all cases. If the function returns false, then the effect on parameter m is unspecified except that m.size() returns 0 and m.empty() returns true. Otherwise the effects on parameter m are given in Table 143.

Element	Value		
m.size()	1 + e.mark_count()		
m.empty()	false		
m.prefix().first	first		
m.prefix().second	m[0].first		
m.prefix().matched	<pre>m.prefix().first != m.prefix().second</pre>		
m.suffix().first	suffix().first m[0].second		
m.suffix().second	last		
m.suffix().matched	<pre>m.suffix().first != m.suffix().second</pre>		
m[0].first	The start of the sequence of characters that matched the		
	regular expression		
m[0].second	The end of the sequence of characters that matched the		
	regular expression		
m[0].matched	true if a match was found, and false otherwise.		
m[n].first	For all integers n < m.size(), the start of the sequence		
	that matched sub-expression n. Alternatively, if sub-		
	expression n did not participate in the match, then last.		
m[n].second	, , ,		
	matched sub-expression n. Alternatively, if sub-expression		
	n did not participate in the match, then last.		
m[n].matched For all integers n < m.size(), true if sub-expres			
	participated in the match, false otherwise.		

Table 143 — Effects of regex\_search algorithm

```
template <class charT, class Allocator, class traits>
   bool regex_search(const charT* str, match_results<const charT*, Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                       regex_constants::match_default);
        Returns: The result of regex_search(str, str + char_traits<charT>::length(str), m, e, flags).
   template <class ST, class SA, class Allocator, class charT, class traits>
     bool regex_search(const basic_string<charT, ST, SA>& s,
                       match_results<
                         typename basic_string<charT, ST, SA>::const_iterator,
                         Allocator>& m,
                       const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type flags =
                         regex_constants::match_default);
        Returns: The result of 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);
        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);
        Returns: regex_search(s.begin(), s.end(), e, flags).
                                                                                      [re.alg.replace]
   28.11.4 regex_replace
   template <class OutputIterator, class BidirectionalIterator,</pre>
       class traits, class charT, class ST, class SA>
     OutputIterator
    regex_replace(OutputIterator out,
                   BidirectionalIterator first, BidirectionalIterator last,
                   const basic_regex<charT, traits>& e,
```

5

7

```
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 =
```

2

3

4

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).

6 Returns: result.

## 28.12 Regular expression iterators

[re.iter]

## 28.12.1 Class template regex\_iterator

[re.regiter]

The class template regex\_iterator is an iterator 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 value a const match\_results<Bidirec

```
namespace std {
 template <class BidirectionalIterator,
            class charT = typename iterator_traits<</pre>
              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;
                                                   pointer;
     typedef const value_type*
     typedef const value_type&
                                                   reference;
     typedef std::forward_iterator_tag
                                                   iterator_category;
     regex_iterator();
     regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                    const regex_type& re,
                    regex_constants::match_flag_type m =
                      regex_constants::match_default);
     regex_iterator(const regex_iterator&);
     regex_iterator& operator=(const regex_iterator&);
     bool operator==(const regex_iterator&) const;
     bool operator!=(const regex_iterator&) const;
```

§ 28.12.1 1087

```
const value_type& operator*() const;
          const value_type* operator->() const;
          regex_iterator& operator++();
          regex_iterator operator++(int);
       private:
          // these members are shown for exposition only:
          BidirectionalIterator
                                                begin;
          BidirectionalIterator
                                                end;
          const regex_type*
                                                pregex;
          regex_constants::match_flag_type
                                                flags;
          match_results<BidirectionalIterator> match;
       };
     }
<sup>2</sup> An object of type regex_iterator that is not an end-of-sequence iterator holds a zero-length match if
   match[0].matched == true and match[0].first == match[0].second. [Note: For example, this can
   occur when the part of the regular expression that matched consists only of an assertion (such as ',', '$',
   '\b', '\B'). — end note
                                                                                       [re.regiter.cnstr]
   28.12.1.1 regex_iterator constructors
   regex_iterator();
        Effects: Constructs an end-of-sequence iterator.
   regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                  const regex_type& re,
                  regex_constants::match_flag_type m = regex_constants::match_default);
2
        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
                                                                                       [re.regiter.comp]
   bool operator==(const regex_iterator& right) const;
1
        Returns: true if *this and right are both end-of-sequence iterators or if the following conditions all
        hold:
          — begin == right.begin,
          — end == right.end,
          — pregex == right.pregex,
          — flags == right.flags, and
          — match[0] == right.match[0];
        otherwise false.
   bool operator!=(const regex_iterator& right) const;
2
        Returns: !(*this == right).
```

§ 28.12.1.2

```
28.12.1.3 regex_iterator dereference
```

[re.regiter.deref]

```
const value_type& operator*() const;

Returns: match.

const value_type* operator->() const;

Returns: &match.
```

#### 28.12.1.4 regex\_iterator increment

[re.regiter.incr]

regex\_iterator& operator++();

- Effects: Constructs a local variable start of type BidirectionalIterator and initializes it with the value of match[0].second.
- If the iterator holds a zero-length match and start == end the operator sets \*this to the end-of-sequence iterator and returns \*this.
- Otherwise, if the iterator holds a zero-length match the operator calls regex\_search(start, end, match, \*pregex, flags | regex\_constants::match\_not\_null | regex\_constants::match\_continuous). If the call returns true the operator returns \*this. Otherwise the operator increments start and continues as if the most recent match was not a zero-length match.
- If the most recent match was not a zero-length match, the operator sets flags to flags | regex\_-constants ::match\_prev\_avail and calls regex\_search(start, end, match, \*pregex, flags). If the call returns false the iterator sets \*this to the end-of-sequence iterator. The iterator then returns \*this.
- In all cases in which the call to regex\_search returns true, match.prefix().first shall be equal to the previous value of match[0].second, and for each index i in the half-open range [0, match.size()) for which match[i].matched is true, match[i].position() shall return distance(begin, match[i].first).
- [Note: This means that match[i].position() gives the offset from the beginning of the target sequence, which is often not the same as the offset from the sequence passed in the call to regex\_search.

   end note]
- <sup>7</sup> It is unspecified how the implementation makes these adjustments.
- [ Note: This means that a compiler may call an implementation-specific search function, in which case a user-defined specialization of  $regex_search$  will not be called. end note ]

#### 28.12.2 Class template regex\_token\_iterator

[re.tokiter]

The class template regex\_token\_iterator is an iterator adaptor; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more sub-expressions for each match found. Each position enumerated by the iterator is a sub\_match class template instance that represents what matched a particular sub-expression within the regular expression.

§ 28.12.2 1089

When class regex\_token\_iterator is used to enumerate a single sub-expression with index -1 the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

- <sup>3</sup> After it is constructed, the iterator finds and stores a value regex\_iterator<BidirectionalIterator> position and sets the internal count N to zero. It also maintains a sequence subs which contains a list of the sub-expressions which will be enumerated. Every time operator++ is used the count N is incremented; if N exceeds or equals subs.size(), then the iterator increments member position and sets count N to zero.
- <sup>4</sup> If the end of sequence is reached (position is equal to the end of sequence iterator), the iterator becomes equal to the end-of-sequence iterator value, unless the sub-expression being enumerated has index -1, in which case the iterator enumerates one last sub-expression that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty sub-expression.
- The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator\* on an end-of-sequence iterator is not defined. For any other iterator value a const sub\_match<BidirectionalIterator>& is returned. The result of operator-> on an end-of-sequence iterator is not defined. For any other iterator value a const sub\_match<BidirectionalIterator>\* is returned.
- It is impossible to store things into regex\_token\_iterators. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```
namespace std {
  template <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);
    template <std::size_t N>
      regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                        const regex_type& re,
```

§ 28.12.2

```
const int (&submatches)[N],
                        regex_constants::match_flag_type m =
                          regex_constants::match_default);
   regex_token_iterator(const regex_token_iterator&);
    regex_token_iterator& operator=(const regex_token_iterator&);
   bool operator==(const regex_token_iterator&) const;
   bool operator!=(const regex_token_iterator&) const;
    const value_type& operator*() const;
    const value_type* operator->() const;
   regex_token_iterator& operator++();
   regex_token_iterator operator++(int);
 private: // data members for exposition only:
    typedef regex_iterator<BidirectionalIterator, charT, traits> position_iterator;
    position_iterator position;
    const value_type *result;
   value_type suffix;
   std::size_t N;
   std::vector<int> subs;
 };
}
```

- <sup>7</sup> 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.
- <sup>8</sup> [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]
- 9 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.

regex\_token\_iterator();

1

[re.tokiter.cnstr]

```
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);
template <std::size_t N>
 regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
```

§ 28.12.2.1

```
const regex_type& re,
const int (&submatches)[N],
regex_constants::match_flag_type m =
  regex_constants::match_default);
```

2 Requires: Each of the initialization values of submatches shall be >= -1.

- Effects: The first constructor initializes the member subs to hold the single value submatch. The second constructor initializes the member subs to hold a copy of the argument submatches. The third and fourth constructors initialize the member subs to hold a copy of the sequence of integer values pointed to by the iterator range [submatches.begin(), submatches.end()) and [&submatches, &submatches + N), respectively.
- Each constructor then sets N to 0, and position to position\_iterator(a, b, re, m). If position is not an end-of-sequence iterator the constructor sets result to the address of the current match. Otherwise if any of the values stored in subs is equal to -1 the constructor sets \*this to a suffix iterator that points to the range [a,b), otherwise the constructor sets \*this to an end-of-sequence iterator.

#### 28.12.2.2 regex\_token\_iterator comparisons

[re.tokiter.comp]

bool operator==(const regex\_token\_iterator& right) const;

Returns: true if \*this and right are both end-of-sequence iterators, or if \*this and right are both suffix iterators and suffix == right.suffix; otherwise returns false if \*this or right is an end-of-sequence iterator or a suffix iterator. Otherwise returns true if position == right.position, N == right.N, and subs == right.subs. Otherwise returns false.

bool operator!=(const regex\_token\_iterator& right) const;

2 Returns: !(\*this == right).

1

2

#### 28.12.2.3 regex\_token\_iterator dereference

[re.tokiter.deref]

```
const value_type& operator*() const;
   Returns: *result.

const value_type* operator->() const;
```

Returns: result.

28.12.2.4 regex\_token\_iterator increment

[re.tokiter.incr]

```
regex_token_iterator& operator++();
```

- Effects: Constructs a local variable prev of type position\_iterator, initialized with the value of position.
- If \*this is a suffix iterator, sets \*this to an end-of-sequence iterator.
- Otherwise, if N + 1 < subs.size(), increments N and sets result to the address of the current match.
- Otherwise, sets N to 0 and increments position. If position is not an end-of-sequence iterator the operator sets result to the address of the current match.
- Otherwise, if any of the values stored in subs is equal to -1 and prev->suffix().length() is not 0 the operator sets \*this to a suffix iterator that points to the range [prev->suffix().first, prev->suffix().second).

§ 28.12.2.4 1092

```
Otherwise, sets *this to an end-of-sequence iterator.

Returns: *this

regex_token_iterator& operator++(int);

Effects: Constructs a copy tmp of *this, then calls ++(*this).

Returns: tmp.
```

## 28.13 Modified ECMAScript regular expression grammar

[re.grammar]

- The regular expression grammar recognized by basic\_regex objects constructed with the ECMAScript flag is that specified by ECMA-262, except as specified below.
- Objects of type specialization of basic\_regex store within themselves a default-constructed instance of their traits template parameter, henceforth referred to as traits\_inst. This traits\_inst object is used to support localization of the regular expression; basic\_regex member functions shall not call any locale dependent C or C++ API, including the formatted string input functions. Instead they shall call the appropriate traits member function to achieve the required effect.
- 3 The following productions within the ECMAScript grammar are modified as follows:

```
ClassAtom ::
-
ClassAtomNoDash
ClassAtomExClass
ClassAtomCollatingElement
ClassAtomEquivalence
```

<sup>4</sup> The following new productions are then added:

```
ClassAtomExClass ::
    [: ClassName :]

ClassAtomCollatingElement ::
    [. ClassName .]

ClassAtomEquivalence ::
    [= ClassName =]

ClassName ::
    ClassName Character
    ClassNameCharacter ClassName

ClassNameCharacter ::
    SourceCharacter but not one of "." "=" ":"
```

- <sup>5</sup> The productions ClassAtomExClass, ClassAtomCollatingElement and ClassAtomEquivalence provide functionality equivalent to that of the same features in regular expressions in POSIX.
- The regular expression grammar may be modified by any regex\_constants::syntax\_option\_type flags specified when constructing an object of type specialization of basic\_regex according to the rules in Table 138.
- A ClassName production, when used in ClassAtomExClass, is not valid if traits\_inst.lookup\_classname returns zero for that name. The names recognized as valid ClassNames are determined by the type of the traits class, but at least the following names shall be recognized: alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit, d, s, w. In addition the following expressions shall be equivalent:

§ 28.13 1093

```
\d and [[:digit:]]
\D and [^[:digit:]]
\s and [[:space:]]
\S and [^[:space:]]
\w and [_[:alnum:]]
\W and [^_[:alnum:]]
```

A ClassName production when used in a ClassAtomCollatingElement production is not valid if the value returned by traits\_inst.lookup\_collatename for that name is an empty string.

- <sup>9</sup> 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.
- 12 If the CV of a UnicodeEscapeSequence is greater than the largest value that can be held in an object of type charT the translator shall throw an exception object of type regex\_error. [Note: This means that values of the form "uxxxx" that do not fit in a character are invalid. end note]
- Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling traits\_inst.value.
- The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMA-262. The behavior is modified according to any match\_flag\_type flags 28.5.2 specified when using the regular expression object in one of the regular expression algorithms 28.11. The behavior is also localized by interaction with the traits class template parameter as follows:
  - 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:

```
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);
```

§ 28.13 1094

— During matching of a regular expression finite state machine against a sequence of characters, testing whether a collating element is a member of a primary equivalence class is conducted by first converting the collating element and the equivalence class to sort keys using traits::transform\_primary, and then comparing the sort keys for equality.

— During matching of a regular expression finite state machine against a sequence of characters, a character c is a member of a character class designated by an iterator range [first,last) if traits\_inst.isctype(c, traits\_inst.lookup\_classname(first, last, flags() & icase)) is true.

§ 28.13 1095

## 29 Atomic operations library

# [atomics]

29.1 General [atomics.general]

This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.

The following subclauses describe atomics requirements and components for types and operations, as summarized below.

Table 144 — Atomics library summary

	Subclause	Header(s)
29.3	Order and Consistency	
29.4	Lock-free Property	
29.5	Atomic Types	<atomic></atomic>
29.6	Operations on Atomic Types	
29.7	Flag Type and Operations	
29.8	Fences	

## 29.2 Header <atomic> synopsis

[atomics.syn]

```
namespace std {
  // 29.3, order and consistency
  enum memory_order;
  template <class T>
    T kill_dependency(T y) noexcept;
  // 29.4, lock-free property
  #define ATOMIC_BOOL_LOCK_FREE unspecified
  #define ATOMIC_CHAR_LOCK_FREE unspecified
  \verb|#define ATOMIC_CHAR16_T_LOCK_FREE unspecified|\\
  #define ATOMIC_CHAR32_T_LOCK_FREE unspecified
  \verb|#define ATOMIC_WCHAR_T_LOCK_FREE unspecified|\\
  #define ATOMIC_SHORT_LOCK_FREE unspecified
  #define ATOMIC_INT_LOCK_FREE unspecified
  #define ATOMIC_LONG_LOCK_FREE unspecified
  #define ATOMIC_LLONG_LOCK_FREE unspecified
  #define ATOMIC_POINTER_LOCK_FREE unspecified
  // 29.5, generic types
  template<class T> struct atomic;
  template<> struct atomic<integral>;
  template<class T> struct atomic<T*>;
  // 29.6.1, general operations on atomic types
  // In the following declarations, atomic-type is either
  // atomic<T> or a named base class for T from
  // Table 145 or inferred from Table 146 or from bool.
  // If it is atomic<T>, then the declaration is a template
```

```
// declaration prefixed with template <class T>.
bool atomic_is_lock_free(const volatile atomic-type*) noexcept;
bool atomic_is_lock_free(const atomic-type*) noexcept;
void atomic_init(volatile atomic-type*, T) noexcept;
void atomic_init(atomic-type*, T) noexcept;
void atomic_store(volatile atomic-type*, T) noexcept;
void atomic_store(atomic-type*, T) noexcept;
void atomic_store_explicit(volatile atomic-type*, T, memory_order) noexcept;
void atomic_store_explicit(atomic-type*, T, memory_order) noexcept;
T atomic_load(const volatile atomic-type*) noexcept;
T atomic_load(const atomic-type*) noexcept;
T atomic_load_explicit(const volatile atomic-type*, memory_order) noexcept;
T atomic_load_explicit(const atomic-type*, memory_order) noexcept;
T atomic_exchange(volatile atomic-type*, T) noexcept;
T atomic_exchange(atomic-type*, T) noexcept;
T atomic_exchange_explicit(volatile atomic-type*, T, memory_order) noexcept;
T atomic_exchange_explicit(atomic-type*, T, memory_order) noexcept;
bool atomic_compare_exchange_weak(volatile atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_weak(atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_strong(volatile atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_strong(atomic-type*, T*, T) noexcept;
bool atomic_compare_exchange_weak_explicit(volatile atomic-type*, T*, T,
  memory_order, memory_order) noexcept;
bool atomic_compare_exchange_weak_explicit(atomic-type*, T*, T.
  memory_order, memory_order) noexcept;
bool atomic_compare)exchange_strong_explicit(volatile atomic-type*, T*, T,
  memory_order, memory_order) noexcept;
bool atomic_compare_exchange_strong_explicit(atomic-type*, T*, T,
  memory_order, memory_order) noexcept;
// 29.6.2, templated operations on atomic types
template <class T>
  T atomic_fetch_add(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_add(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_add_explicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_add_explicit(atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_sub(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_sub(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_sub_explicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_sub_explicit(atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_and(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_and(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_and_explicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_and_explicit(atomic<T>*, T, memory_order) noexcept;
```

```
template <class T>
  T atomic_fetch_or(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_or(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_or_explicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_or_explicit(atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_xor(volatile atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_xor(atomic<T>*, T) noexcept;
template <class T>
  T atomic_fetch_xor_explicit(volatile atomic<T>*, T, memory_order) noexcept;
template <class T>
  T atomic_fetch_xor_explicit(atomic<T>*, T, memory_order) noexcept;
// 29.6.3, arithmetic operations on atomic types
// In the following declarations, atomic-integral is either
// atomic<T> or a named base class for T from
// Table 145 or inferred from Table 146.
// If it is atomic<T>, then the declaration is a template
// specialization declaration prefixed with template <>.
integral atomic_fetch_add(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_add(atomic-integral*, integral) noexcept;
integral atomic_fetch_add_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_add_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_sub(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_sub(atomic-integral*, integral) noexcept;
integral atomic_fetch_sub_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_sub_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_and(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_and(atomic-integral*, integral) noexcept;
integral atomic_fetch_and_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_and_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_or(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_or(atomic-integral*, integral) noexcept;
integral atomic_fetch_or_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_or_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_xor(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_xor(atomic-integral*, integral) noexcept;
integral atomic_fetch_xor_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_xor_explicit(atomic-integral*, integral, memory_order) noexcept;
// 29.6.4, partial specializations for pointers
template <class T>
  T* atomic_fetch_add(volatile atomic<T*>*, ptrdiff_t) noexcept;
template <class T>
  T* atomic_fetch_add(atomic<T*>*, ptrdiff_t) noexcept;
template <class T>
  T* atomic_fetch_add_explicit(volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
template <class T>
  T* atomic_fetch_add_explicit(atomic<T*>*, ptrdiff_t, memory_order) noexcept;
```

```
template <class T>
     T* atomic_fetch_sub(volatile atomic<T*>*, ptrdiff_t) noexcept;
   template <class T>
     T* atomic_fetch_sub(atomic<T*>*, ptrdiff_t) noexcept;
   template <class T>
     T* atomic_fetch_sub_explicit(volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
    template <class T>
     T* atomic_fetch_sub_explicit(atomic<T*>*, ptrdiff_t, memory_order) noexcept;
    // 29.6.5, initialization
    #define ATOMIC_VAR_INIT(value) see below
    // 29.7, flag type and operations
    struct atomic_flag;
    bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
   bool atomic_flag_test_and_set(atomic_flag*) noexcept;
    bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order) noexcept;
    bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order) noexcept;
   void atomic_flag_clear(volatile atomic_flag*) noexcept;
   void atomic_flag_clear(atomic_flag*) noexcept;
    void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order) noexcept;
    void atomic_flag_clear_explicit(atomic_flag*, memory_order) noexcept;
    #define ATOMIC_FLAG_INIT see below
    // 29.8, fences
    extern "C" void atomic_thread_fence(memory_order) noexcept;
    extern "C" void atomic_signal_fence(memory_order) noexcept;
29.3
       Order and consistency
                                                                                   [atomics.order]
 namespace std {
    typedef enum memory_order {
     memory_order_relaxed, memory_order_consume, memory_order_acquire,
     memory_order_release, memory_order_acq_rel, memory_order_seq_cst
    } memory_order;
```

- 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]

An atomic operation A that performs a release operation on an atomic object M synchronizes with an atomic operation B that performs an acquire operation on M and takes its value from any side effect in the release sequence headed by A.

- There shall be a single total order S on all memory\_order\_seq\_cst operations, consistent with the "happens before" order and modification orders for all affected locations, such that each memory\_order\_seq\_cst operation B that loads a value from an atomic object M observes one of the following values:
  - the result of the last modification A of M that precedes B in S, if it exists, or
  - if A exists, the result of some modification of M in the visible sequence of side effects with respect to B that is not memory\_order\_seq\_cst and that does not happen before A, or
  - if A does not exist, the result of some modification of M in the visible sequence of side effects with respect to B that is not memory\_order\_seq\_cst.

[Note: Although it is not explicitly required that S include locks, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the "happens before" ordering. — end note]

- <sup>4</sup> For an atomic operation B that reads the value of an atomic object M, if there is a memory\_order\_seq\_-cst fence X sequenced before B, then B observes either the last memory\_order\_seq\_cst modification of M preceding X in the total order S or a later modification of M in its modification order.
- For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a memory\_order\_seq\_cst fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.
- For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there are memory\_order\_seq\_cst fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B observes either the effects of A or a later modification of M in its modification order.
- For atomic operations A and B on an atomic object M, if there are memory\_order\_seq\_cst fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B occurs later than A in the modification order of M.
- 8 [Note: memory\_order\_seq\_cst ensures sequential consistency only for a program that is free of data races and uses exclusively memory\_order\_seq\_cst operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In particular, memory\_order\_seq\_cst fences ensure a total order only for the fences themselves. Fences cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications. end note]
- <sup>9</sup> An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence. The ordering of evaluations in this sequence shall be such that:
  - if an evaluation B observes a value computed by A in a different thread, then B 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.
- [Note: The second requirement disallows "out-of-thin-air" or "speculative" stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations may appear in this sequence out of thread order. For example, with x and y initially zero,

```
// Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);
```

```
// Thread 2:
     r2 = x.load(memory_order_relaxed);
     y.store(42, memory_order_relaxed);
       is allowed to produce r1 = r2 = 42. The sequence of evaluations justifying this consists of:
     y.store(42, memory_order_relaxed);
     r1 = y.load(memory_order_relaxed);
     x.store(r1, memory_order_relaxed);
     r2 = x.load(memory_order_relaxed);
       On the other hand,
      // Thread 1:
     r1 = y.load(memory_order_relaxed);
     x.store(r1, memory_order_relaxed);
      // Thread 2:
     r2 = x.load(memory_order_relaxed);
     y.store(r2, memory_order_relaxed);
       may not produce r1 = r2 = 42, since there is no sequence of evaluations that results in the computation
   of 42. In the absence of "relaxed" operations and read-modify-write operations with weaker than memory_-
   order_acq_rel ordering, the second requirement has no impact. — end note]
   [Note: The requirements do allow r1 == r2 == 42 in the following example, with x and y initially zero:
     // Thread 1:
     r1 = x.load(memory_order_relaxed);
     if (r1 == 42) y.store(r1, memory_order_relaxed);
      // Thread 2:
     r2 = y.load(memory_order_relaxed);
      if (r2 == 42) x.store(42, memory_order_relaxed);
       However, implementations should not allow such behavior. — end note
12 Atomic read-modify-write operations shall always read the last value (in the modification order) written
   before the write associated with the read-modify-write operation.
   Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.
   template <class T>
     T kill_dependency(T y) noexcept;
14
         Effects: The argument does not carry a dependency to the return value (1.10).
15
         Returns: y.
   29.4
           Lock-free property
                                                                                      [atomics.lockfree]
      #define ATOMIC_BOOL_LOCK_FREE unspecified
      #define ATOMIC_CHAR_LOCK_FREE unspecified
      #define ATOMIC_CHAR16_T_LOCK_FREE unspecified
      #define ATOMIC_CHAR32_T_LOCK_FREE unspecified
      #define ATOMIC_WCHAR_T_LOCK_FREE unspecified
      #define ATOMIC_SHORT_LOCK_FREE unspecified
      #define ATOMIC_INT_LOCK_FREE unspecified
      #define ATOMIC_LONG_LOCK_FREE unspecified
      #define ATOMIC_LLONG_LOCK_FREE unspecified
      #define ATOMIC_POINTER_LOCK_FREE unspecified
```

The ATOMIC\_...\_LOCK\_FREE macros indicate the lock-free property of the corresponding atomic types, with the signed and unsigned variants grouped together. The properties also apply to the corresponding (partial) specializations of the atomic template. A value of 0 indicates that the types are never lock-free. A value of 1 indicates that the types are 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 by memory that is mapped into a process more than once and by memory that is shared between two processes. end note

## 29.5 Atomic types

[atomics.types.generic]

```
namespace std {
  template <class T> struct atomic {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T, memory_order = memory_order_seq_cst) volatile noexcept;
    void store(T, memory_order = memory_order_seq_cst) noexcept;
    T load(memory_order = memory_order_seq_cst) const volatile noexcept;
    T load(memory_order = memory_order_seq_cst) const noexcept;
    operator T() const volatile noexcept;
    operator T() const noexcept;
    T exchange(T, memory_order = memory_order_seq_cst) volatile noexcept;
    T exchange(T, memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(T&, T, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(T&, T, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(T&, T, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_strong(T&, T, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst) noexcept;
    atomic() noexcept = default;
    constexpr atomic(T) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    T operator=(T) volatile noexcept;
    T operator=(T) noexcept;
  };
 template <> struct atomic<integral> {
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    void store(integral, memory_order = memory_order_seq_cst) noexcept;
    integral load(memory_order = memory_order_seq_cst) const volatile noexcept;
    integral load(memory_order = memory_order_seq_cst) const noexcept;
    operator integral() const volatile noexcept;
    operator integral() const noexcept;
    integral exchange(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    integral exchange(integral, memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(integral&, integral, memory_order, memory_order) volatile noexcept;
```

```
bool compare_exchange_weak(integral&, integral, memory_order, memory_order) noexcept;
  bool compare_exchange_strong(integral &, integral, memory_order, memory_order) volatile noexcept;
 bool compare_exchange_strong(integral &, integral, memory_order, memory_order) noexcept;
  bool compare_exchange_weak(integral &, integral, memory_order = memory_order_seq_cst) volatile noexcept;
 bool compare_exchange_weak(integral &, integral , memory_order = memory_order_seq_cst) noexcept;
 bool compare_exchange_strong(integral &, integral, memory_order = memory_order_seq_cst) volatile noexcept;
  bool compare exchange strong(integral&, integral, memory_order = memory_order_seq_cst) noexcept;
  integral fetch_add(integral, memory_order = memory_order_seq_cst) volatile noexcept;
  integral fetch_add(integral, memory_order = memory_order_seq_cst) noexcept;
  integral fetch_sub(integral, memory_order = memory_order_seq_cst) volatile noexcept;
  integral fetch_sub(integral, memory_order = memory_order_seq_cst) noexcept;
  integral fetch_and(integral, memory_order = memory_order_seq_cst) volatile noexcept;
  integral fetch_and(integral, memory_order = memory_order_seq_cst) noexcept;
  integral fetch_or(integral, memory_order = memory_order_seq_cst) volatile noexcept;
  integral fetch_or(integral, memory_order = memory_order_seq_cst) noexcept;
  integral fetch_xor(integral, memory_order = memory_order_seq_cst) volatile noexcept;
  integral fetch_xor(integral, memory_order = memory_order_seq_cst) noexcept;
  atomic() noexcept = default;
  constexpr atomic(integral) noexcept;
  atomic(const atomic&) = delete;
  atomic& operator=(const atomic&) = delete;
  atomic& operator=(const atomic&) volatile = delete;
  integral operator=(integral) volatile noexcept;
  integral operator=(integral) noexcept;
  integral operator++(int) volatile noexcept;
  integral operator++(int) noexcept;
  integral operator--(int) volatile noexcept;
  integral operator--(int) noexcept;
  integral operator++() volatile noexcept;
  integral operator++() noexcept;
  integral operator--() volatile noexcept;
  integral operator--() noexcept;
  integral operator+=(integral) volatile noexcept;
  integral operator+=(integral) noexcept;
  integral operator-=(integral) volatile noexcept;
  integral operator==(integral) noexcept;
  integral operator&=(integral) volatile noexcept;
  integral operator&=(integral) noexcept;
  integral operator|=(integral) volatile noexcept;
  integral operator|=(integral) noexcept;
  integral operator^=(integral) volatile noexcept;
  integral operator^=(integral) noexcept;
};
template <class T> struct atomic<T*> {
 bool is_lock_free() const volatile noexcept;
 bool is_lock_free() const noexcept;
 void store(T*, memory_order = memory_order_seq_cst) volatile noexcept;
 void store(T*, memory_order = memory_order_seq_cst) noexcept;
 T* load(memory_order = memory_order_seq_cst) const volatile noexcept;
 T* load(memory_order = memory_order_seq_cst) const noexcept;
  operator T*() const volatile noexcept;
  operator T*() const noexcept;
```

§ 29.5

```
T* exchange(T*, memory_order = memory_order_seq_cst) volatile noexcept;
   T* exchange(T*, memory_order = memory_order_seq_cst) noexcept;
   bool compare_exchange_weak(T*&, T*, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order, memory_order) noexcept;
   bool compare_exchange_strong(T*&, T*, memory_order, memory_order) volatile noexcept;
   bool compare_exchange_strong(T*&, T*, memory_order, memory_order) noexcept;
   bool compare_exchange_weak(T*&, T*, memory_order = memory_order_seq_cst) volatile noexcept;
   bool compare exchange weak(T*&, T*, memory order = memory order seq cst) noexcept;
   bool compare exchange_strong(T*&, T*, memory_order = memory_order seq_cst) volatile noexcept;
   bool compare_exchange_strong(T*&, T*, memory_order = memory_order_seq_cst) noexcept;
   T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
   T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
   T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
    T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
    atomic() noexcept = default;
    constexpr atomic(T*) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    T* operator=(T*) volatile noexcept;
   T* operator=(T*) noexcept;
    T* operator++(int) volatile noexcept;
   T* operator++(int) noexcept;
   T* operator--(int) volatile noexcept;
   T* operator--(int) noexcept;
   T* operator++() volatile noexcept;
   T* operator++() noexcept;
   T* operator--() volatile noexcept;
   T* operator--() noexcept;
   T* operator+=(ptrdiff_t) volatile noexcept;
   T* operator+=(ptrdiff_t) noexcept;
   T* operator-=(ptrdiff_t) volatile noexcept;
   T* operator-=(ptrdiff_t) noexcept;
 };
}
```

- 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]
- <sup>2</sup> The semantics of the operations on specializations of atomic are defined in 29.6.
- <sup>3</sup> Specializations and instantiations of the atomic template shall have a deleted copy constructor, a deleted copy assignment operator, and a constexpr value constructor.
- There shall be full specializations of the atomic template for the integral types char, signed char, unsigned char, unsigned char, unsigned long, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, char16\_t, char32\_t, wchar\_t, and any other types needed by the typedefs in the header <cstdint>. For each integral type integral, the specialization atomic<integral> provides additional atomic operations appropriate to integral types. There shall be a specialization atomic<br/>
  specialization
- <sup>5</sup> The atomic integral specializations and the specialization atomic<bool> shall have standard layout. They shall each have a trivial default constructor and a trivial destructor. They shall each support aggregate initialization syntax.
- <sup>6</sup> There shall be pointer partial specializations of the atomic class template. These specializations shall have

§ 29.5

standard layout, trivial default constructors, and trivial destructors. They shall each support aggregate initialization syntax.

There shall be named types corresponding to the integral specializations of atomic, as specified in Table 145, and a named type atomic\_bool corresponding to the specified atomic<br/>bool>. Each named type is a either typedef to the corresponding specialization or a base class of the corresponding specialization. If it is a base class, it shall support the same member functions as the corresponding specialization.

Named type	Integral argument type
atomic_char	char
atomic_schar	signed char
atomic_uchar	unsigned char
atomic_short	short
atomic_ushort	unsigned short
atomic_int	int
atomic_uint	unsigned int
atomic_long	long
atomic_ulong	unsigned long
atomic_llong	long long
atomic_ullong	unsigned long long
atomic_char16_t	char16_t
atomic_char32_t	char32_t
atomic_wchar_t	wchar_t

Table 145 — atomic integral typedefs

- There shall be atomic typedefs corresponding to the typedefs in the header <inttypes.h> as specified in Table 146.
- [Note: The representation of an atomic specialization need not have the same size as its corresponding argument type. Specializations should have the same size whenever possible, as this reduces the effort required to port existing code. end note]

#### 29.6 Operations on atomic types

#### [atomics.types.operations]

## 29.6.1 General operations on atomic types

[atomics.types.operations.general]

- <sup>1</sup> The implementation shall provide the functions and function templates identified as "general operations on atomic types" in 29.2.
- <sup>2</sup> In the declarations of these functions and function templates, the name *atomic-type* refers to either atomic<T> or to a named base class for T from Table 145 or inferred from Table 146.

# 29.6.2 Templated operations on atomic types [atomics.types.operations.templ]

The implementation shall declare but not define the function templates identified as "templated operations on atomic types" in 29.2.

## 29.6.3 Arithmetic operations on atomic types [atomics.types.operations.arith]

- <sup>1</sup> The implementation shall provide the functions and function template specializations identified as "arithmetic operations on atomic types" in 29.2.
- In the declarations of these functions and function template specializations, the name integral refers to an integral type and the name atomic-integral refers to either atomic<integral> or to a named base class for integral from Table 145 or inferred from Table 146.

# 29.6.4 Operations on atomic pointer types [atomics.types.operations.pointer]

<sup>1</sup> The implementation shall provide the function template specializations identified as "partial specializations

Table 146 — atomic <inttypes.h> typedefs

Atomic typedef	<pre><inttypes.h> type</inttypes.h></pre>	
atomic_int_least8_t	int_least8_t	
atomic_uint_least8_t	uint_least8_t	
atomic_int_least16_t	int_least16_t	
atomic_uint_least16_t	uint_least16_t	
atomic_int_least32_t	int_least32_t	
atomic_uint_least32_t	uint_least32_t	
atomic_int_least64_t	int_least64_t	
atomic_uint_least64_t	uint_least64_t	
atomic_int_fast8_t	int_fast8_t	
atomic_uint_fast8_t	uint_fast8_t	
atomic_int_fast16_t	int_fast16_t	
atomic_uint_fast16_t	uint_fast16_t	
atomic_int_fast32_t	int_fast32_t	
atomic_uint_fast32_t	uint_fast32_t	
atomic_int_fast64_t	int_fast64_t	
atomic_uint_fast64_t	uint_fast64_t	
atomic_intptr_t	intptr_t	
atomic_uintptr_t	uintptr_t	
atomic_size_t	size_t	
atomic_ptrdiff_t	ptrdiff_t	
atomic_intmax_t	intmax_t	
atomic_uintmax_t	uintmax_t	

for pointers" in 29.2.

## 29.6.5 Requirements for operations on atomic types [atomics.types.operations.req]

- There are only a few kinds of operations on atomic types, though there are many instances on those kinds. This section specifies each general kind. The specific instances are defined in 29.5, 29.6.1, 29.6.3, and 29.6.4.
- <sup>2</sup> 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  $\mathtt{std}$ ::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.
- 3 [Note: Many operations are volatile-qualified. The "volatile as device register" semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects. It does not mean that operations on non-volatile objects become volatile. Thus, volatile qualified operations on non-volatile objects may be merged under some conditions. end note

#### A::A() noexcept = default;

Effects: leaves the atomic object in an uninitialized state. [Note: These semantics ensure compatibility with C. — end note]

```
constexpr A::A(C desired) noexcept;
```

Effects: Initializes the object with the value desired. Initialization is not an atomic operation (1.10). [Note: it is possible to have an access to an atomic object A race with its construction, for example by communicating the address of the just-constructed object A to another thread via memory\_order\_relaxed operations on a suitable atomic pointer variable, and then immediately accessing A in the receiving thread. This results in undefined behavior. —end note]

#define ATOMIC\_VAR\_INIT(value) see below

7

11

12

The macro expands to a token sequence suitable for constant initialization of an atomic variable of static storage duration of a type that is initialization-compatible with *value*. [Note: This operation may need to initialize locks. — end note] Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race. [Example:

```
atomic<int> v = ATOMIC_VAR_INIT(5);

— end example]

bool atomic_is_lock_free(const volatile A *object) noexcept;
bool atomic_is_lock_free(const A *object) noexcept;
bool A::is_lock_free() const volatile noexcept;
bool A::is_lock_free() const noexcept;

Returns: True if the object's operations are lock-free, false otherwise.
```

```
void atomic_init(volatile A *object, C desired) noexcept;
void atomic_init(A *object, C desired) noexcept;
```

Effects: Non-atomically initializes \*object with value desired. This function shall only be applied to objects that have been default constructed, and then only once. [Note: These semantics ensure compatibility with C. — end note] [Note: Concurrent access from another thread, even via an atomic operation, constitutes a data race. — end note]

```
void atomic_store(volatile A* object, C desired) noexcept;
void atomic_store(A* object, C desired) noexcept;
void atomic_store_explicit(volatile A *object, C desired, memory_order order) noexcept;
void atomic_store_explicit(A* object, C desired, memory_order order) noexcept;
void A::store(C desired, memory_order order = memory_order_seq_cst) volatile noexcept;
void A::store(C desired, memory_order order = memory_order_seq_cst) noexcept;
```

Requires: The order argument shall not be memory\_order\_consume, memory\_order\_acquire, nor memory\_order\_acq\_rel.

Effects: Atomically replaces the value pointed to by object or by this with the value of desired.

Memory is affected according to the value of order.

```
C A::operator=(C desired) volatile noexcept;
C A::operator=(C desired) noexcept;

Effects: store(desired)

Returns: desired
```

```
C atomic_load(const volatile A* object) noexcept;
   C atomic_load(const A* object) noexcept;
   {\it C} atomic_load_explicit(const volatile {\it A*} object, memory_order) noexcept;
   C atomic_load_explicit(const A* object, memory_order) noexcept;
   C A::load(memory_order order = memory_order_seq_cst) const volatile noexcept;
    C A::load(memory_order order = memory_order_seq_cst) const noexcept;
13
         Requires: The order argument shall not be memory_order_release nor memory_order_acq_rel.
14
         Effects: Memory is affected according to the value of order.
15
         Returns: Atomically returns the value pointed to by object or by this.
   A::operator C() const volatile noexcept;
    A::operator C() const noexcept;
16
         Effects: load()
17
         Returns: The result of load().
   C atomic_exchange(volatile A* object, C desired) noexcept;
   C atomic_exchange(A* object, C desired) noexcept;
   C atomic_exchange_explicit(volatile A* object, C desired, memory_order) noexcept;
   C atomic_exchange_explicit(A* object, C desired, memory_order) noexcept;
   C A::exchange(C desired, memory_order order = memory_order_seq_cst) volatile noexcept;
   C A::exchange(C desired, memory_order order = memory_order_seq_cst) noexcept;
18
         Effects: Atomically replaces the value pointed to by object or by this with desired. Memory
         is affected according to the value of order. These operations are atomic read-modify-write opera-
         tions (1.10).
19
         Returns: Atomically returns the value pointed to by object or by this immediately before the effects.
   \verb|bool atomic_compare_exchange_weak(volatile A* object, C* expected, C desired) no except; \\
   bool atomic_compare_exchange_weak(A* object, C* expected, C desired) noexcept;
   bool atomic_compare_exchange_strong(volatile A* object, C* expected, C desired) noexcept;
   bool atomic_compare_exchange_strong(A* object, C* expected, C desired) noexcept;
   bool atomic_compare_exchange_weak_explicit(volatile A* object, C* expected, C desired,
       memory_order success, memory_order failure) noexcept;
   bool atomic_compare_exchange_weak_explicit(A* object, C* expected, C desired,
       memory_order success, memory_order failure) noexcept;
   bool atomic_compare_exchange_strong_explicit(volatile A* object, C* expected, C desired,
       memory_order success, memory_order failure) noexcept;
   bool atomic_compare_exchange_strong_explicit(A* object, C* expected, C desired,
       memory_order success, memory_order failure) noexcept;
   bool A::compare_exchange_weak(C& expected, C desired,
       memory_order success, memory_order failure) volatile noexcept;
   bool A::compare_exchange_weak(C& expected, C desired,
       memory_order success, memory_order failure) noexcept;
   bool A::compare_exchange_strong(C& expected, C desired,
       memory_order success, memory_order failure) volatile noexcept;
   bool A::compare_exchange_strong(C& expected, C desired,
       memory_order success, memory_order failure) noexcept;
   bool A::compare_exchange_weak(C& expected, C desired,
       memory_order order = memory_order_seq_cst) volatile noexcept;
```

```
bool A::compare_exchange_weak(C& expected, C desired,
    memory_order order = memory_order_seq_cst) noexcept;
bool A::compare_exchange_strong(C& expected, C desired,
    memory_order order = memory_order_seq_cst) volatile noexcept;
bool A::compare_exchange_strong(C& expected, C desired,
    memory_order order = memory_order_seq_cst) noexcept;
```

Requires: The failure argument shall not be memory\_order\_release nor memory\_order\_acq\_rel.

The failure argument shall be no stronger than the success argument.

Effects: Atomically, compares the contents of the memory pointed to by object or by this for equality with that in expected, and if true, replaces the contents of the memory pointed to by object or by this with that in desired, and if false, updates the contents of the memory in expected with the contents of the memory pointed to by object or by this. Further, if the comparison is true, memory is affected according to the value of success, and if the comparison is false, memory is affected according to the value of failure. When only one memory\_order argument is supplied, the value of success is order, and the value of failure is order except that a value of memory\_order\_acq\_rel shall be replaced by the value memory\_order\_acquire and a value of memory\_order\_release shall be replaced by the value memory\_order\_released. If the operation returns true, these operations are atomic read-modify-write operations (1.10). Otherwise, these operations are atomic load operations.

Returns: The result of the comparison.

21

22

23

24

25

[Note: For example, the effect of atomic\_compare\_exchange\_strong 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 {
  desired = function(expected);
} while (!current.compare_exchange_weak(expected, desired));

— end example]
```

Implementations should ensure that weak compare-and-exchange operations do not consistently return false unless either the atomic object has value different from expected or there are concurrent modifications to the atomic object.

Remark: A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by expected and object are equal, it may return false and store back to expected the same memory contents that were originally there. [Note: This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines. A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop.

When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable.  $-end\ note$ 

[Note: The memcpy and memcmp semantics of the compare-and-exchange operations may result in failed comparisons for values that compare equal with operator== if the underlying type has padding bits,

trap bits, or alternate representations of the same value. Thus,  $compare_exchange_strong$  should be used with extreme care. On the other hand,  $compare_exchange_weak$  should converge rapidly. — end note

<sup>27</sup> The following operations perform arithmetic computations. The key, operator, and computation correspondence is:

Key	Op	Computation	Key	Op	Computation
add	+	addition	sub	-	subtraction
or		bitwise inclusive or	xor	^	bitwise exclusive or
and	&	bitwise and			

```
C atomic_fetch_key(volatile A *object, M operand) noexcept;
C atomic_fetch_key(A* object, M operand) noexcept;
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order) noexcept;
C atomic_fetch_key_explicit(A* object, M operand, memory_order order) noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) volatile noexcept;
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) noexcept;
```

Effects: Atomically replaces the value pointed to by object or by this with the result of the *computation* applied to the value pointed to by object or by this and the given operand. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (1.10).

29 Returns: Atomically, the value pointed to by object or by this immediately before the effects.

*Remark:* For signed integer types, arithmetic is defined to use two's complement representation. There are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

```
C A::operator op=(M operand) volatile noexcept;
   C A::operator op=(M operand) noexcept;
31
         Effects: fetch_key(operand)
32
         Returns: fetch_key(operand) op operand
   C A::operator++(int) volatile noexcept;
   C A::operator++(int) noexcept;
33
         Returns: fetch_add(1)
   C A::operator--(int) volatile noexcept;
   C A::operator--(int) noexcept;
34
         Returns: fetch_sub(1)
   C A::operator++() volatile noexcept;
   C A::operator++() noexcept;
```

30

```
35  Effects: fetch_add(1)
36  Returns: fetch_add(1) + 1

C A::operator--() volatile noexcept;
C A::operator--() noexcept;

37  Effects: fetch_sub(1)
38  Returns: fetch_sub(1) - 1
```

## 29.7 Flag type and operations

[atomics.flag]

```
namespace std {
 typedef struct atomic_flag {
    bool test_and_set(memory_order = memory_order_seq_cst) volatile noexcept;
    bool test_and_set(memory_order = memory_order_seq_cst) noexcept;
    void clear(memory_order = memory_order_seq_cst) volatile noexcept;
    void clear(memory_order = memory_order_seq_cst) noexcept;
    atomic_flag() noexcept = default;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) volatile = delete;
  } atomic_flag;
  bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
  bool atomic_flag_test_and_set(atomic_flag*) noexcept;
  bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order) noexcept;
  bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order) noexcept;
 void atomic_flag_clear(volatile atomic_flag*) noexcept;
  void atomic_flag_clear(atomic_flag*) noexcept;
  void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear_explicit(atomic_flag*, memory_order) noexcept;
  #define ATOMIC_FLAG_INIT see below
}
```

- The atomic\_flag type provides the classic test-and-set functionality. It has two states, set and clear.
- Operations on an object of type atomic\_flag shall be lock-free. [Note: Hence the operations should also be address-free. No other type requires lock-free operations, so the atomic\_flag type is the minimum hardware-implemented type needed to conform to this International standard. The remaining types can be emulated with atomic\_flag, though with less than ideal properties. end note]
- The atomic\_flag type shall have standard layout. It shall have a trivial default constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor.
- The macro ATOMIC\_FLAG\_INIT shall be defined in such a way that it can be used to initialize an object of type atomic\_flag to the clear state. For a static-duration object, that initialization shall be static. It is unspecified whether an uninitialized atomic\_flag object has an initial state of set or clear. [Example:

```
atomic_flag guard = ATOMIC_FLAG_INIT;

— end example]

bool atomic_flag_test_and_set(volatile atomic_flag *object) noexcept;
bool atomic_flag_test_and_set(atomic_flag *object) noexcept;
bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object, memory_order order) noexcept;
```

§ 29.7

```
bool atomic_flag_test_and_set_explicit(atomic_flag *object, memory_order order) noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst) volatile noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst) noexcept;

Effects: Atomically sets the value pointed to by object or by this to true. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (1.10).
```

6 Returns: Atomically, the value of the object immediately before the effects.

5

```
void atomic_flag_clear(volatile atomic_flag *object) noexcept;
void atomic_flag_clear(atomic_flag *object) noexcept;
void atomic_flag_clear_explicit(volatile atomic_flag *object, memory_order order) noexcept;
void atomic_flag_clear_explicit(atomic_flag *object, memory_order order) noexcept;
void atomic_flag::clear(memory_order order = memory_order_seq_cst) volatile noexcept;
void atomic_flag::clear(memory_order order = memory_order_seq_cst) noexcept;
```

- 7 Requires: The order argument shall not be memory\_order\_acquire or memory\_order\_acq\_rel.
- 8 Effects: Atomically sets the value pointed to by object or by this to false. Memory is affected according to the value of order.

29.8 Fences [atomics.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.

extern "C" void atomic\_signal\_fence(memory\_order order) noexcept;

Effects: equivalent to atomic\_thread\_fence(order), except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.

— is a sequentially consistent acquire and release fence, if order == memory\_order\_seq\_cst.

Note: atomic\_signal\_fence can be used to specify the order in which actions performed by the thread become visible to the signal handler.

§ 29.8 1112

Note: compiler optimizations and reorderings of loads and stores are inhibited in the same way as with atomic\_thread\_fence, but the hardware fence instructions that atomic\_thread\_fence would have inserted are not emitted.

§ 29.8

# 30 Thread support library

[thread]

30.1 General [thread.general]

The following subclauses describe components to create and manage threads (1.10), perform mutual exclusion, and communicate conditions and values between threads, as summarized in Table 148.

	Subclause	Header(s)
30.2	Requirements	
30.3	Threads	<thread></thread>
30.4	Mutual exclusion	<mutex></mutex>
30.5	Condition variables	<pre><condition_variable></condition_variable></pre>
30.6	Futures	<future></future>

# 30.2 Requirements

[thread.req]

## 30.2.1 Template parameter names

[thread.req.paramname]

- <sup>1</sup> Throughout this Clause, the names of template parameters are used to express type requirements.
- <sup>2</sup> 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

# [thread.req.exception]

Some functions described in this Clause are specified to throw exceptions of type system\_error (19.5.6). Such exceptions shall be thrown if any of the function's error conditions is detected or a call to an operating system or other underlying API results in an error that prevents the library function from meeting its specifications. Failure to allocate storage shall be reported as described in 17.6.5.12.

[Example: Consider a function in this clause that is specified to throw exceptions of type system\_error and specifies error conditions that include operation\_not\_permitted for a thread that does not have the privilege to perform the operation. Assume that, during the execution of this function, an errno of EPERM is reported by a POSIX API call used by the implementation. Since POSIX specifies an errno of EPERM when "the caller does not have the privilege to perform the operation", the implementation maps EPERM to an error\_condition of operation\_not\_permitted (19.5) and an exception of type system\_error is thrown.—end example]

<sup>2</sup> 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

[thread.req.native]

Several classes described in this Clause have members native\_handle\_type and native\_handle. The presence of these members and their semantics is implementation-defined. [Note: These members allow implementations to provide access to implementation details. Their names are specified to facilitate portable compile-time detection. Actual use of these members is inherently non-portable. —end note]

#### 30.2.4 Timing specifications

[thread.reg.timing]

- <sup>1</sup> 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 20.11.
- <sup>2</sup> Implementations necessarily have some delay in returning from a timeout. Any overhead in interrupt response, function return, and scheduling induces a "quality of implementation" delay, expressed as duration

§ 30.2.4

 $D_i$ . Ideally, this delay would be zero. Further, any contention for processor and memory resources induces a "quality of management" delay, expressed as duration  $D_m$ . The delay durations may vary from timeout to timeout, but in all cases shorter is better.

- <sup>3</sup> The member functions whose names end in \_for take an argument that specifies a duration. These functions produce relative timeouts. Implementations should use a steady clock to measure time for these functions. <sup>336</sup> Given a duration argument  $D_t$ , the real-time duration of the timeout is  $D_t + D_i + D_m$ .
- The member functions whose names end in \_until take an argument that specifies a time point. These functions produce absolute timeouts. Implementations should use the clock specified in the time point to measure time for these functions. Given a clock time point argument  $C_t$ , the clock time point of the return from timeout should be  $C_t + D_i + D_m$  when the clock is not adjusted during the timeout. If the clock is adjusted to the time  $C_a$  during the timeout, the behavior should be as follows:
  - if  $C_a > C_t$ , the waiting function should wake as soon as possible, i.e.  $C_a + D_i + D_m$ , since the timeout is already satisfied. [Note: This specification may result in the total duration of the wait decreasing when measured against a steady clock. end note]
  - if  $C_a <= C_t$ , the waiting function should not time out until Clock::now() returns a time  $C_n >= C_t$ , i.e. waking at  $C_t + D_i + D_m$ . [Note: When the clock is adjusted backwards, this specification may result in the total duration of the wait increasing when measured against a steady clock. When the clock is adjusted forwards, this specification may result in the total duration of the wait decreasing when measured against a steady clock. —end note]

An implementation shall return from such a timeout at any point from the time specified above to the time it would return from a steady-clock relative timeout on the difference between  $C_t$  and the time point of the call to the <u>\_until</u> function. [Note: Implementations should decrease the duration of the wait when the clock is adjusted forwards. — end note]

- <sup>5</sup> [Note: If the clock is not synchronized with a steady clock, e.g., a CPU time clock, these timeouts might not provide useful functionality. end note]
- The resolution of timing provided by an implementation depends on both operating system and hardware. The finest resolution provided by an implementation is called the *native resolution*.
- <sup>7</sup> Implementation-provided clocks that are used for these functions shall meet the TrivialClock requirements (20.11.3).

# 30.2.5 Requirements for Lockable types

# [thread.req.lockable]

## 30.2.5.1 In general

#### [thread.req.lockable.general]

- An execution agent is an entity such as a thread that may perform work in parallel with other execution agents. [Note: Implementations or users may introduce other kinds of agents such as processes or thread-pool tasks. —end note] The calling agent is determined by context, e.g. the calling thread that contains the call, and so on.
- <sup>2</sup> [Note: Some lockable objects are "agent oblivious" in that they work for any execution agent model because they do not determine or store the agent's ID (e.g., an ordinary spin lock). end note]
- The standard library templates unique\_lock (30.4.2.2), lock\_guard (30.4.2.1), lock, try\_lock (30.4.3), and condition\_variable\_any (30.5.2) all operate on user-supplied lockable objects. The BasicLockable requirements, the Lockable requirements, and the TimedLockable requirements list the requirements imposed by these library types in order to acquire or release ownership of a lock by a given execution agent. [Note: The nature of any lock ownership and any synchronization it may entail are not part of these requirements. end note]

## 30.2.5.2 BasicLockable requirements

## [thread.reg.lockable.basic]

A type L meets the BasicLockable requirements if the following expressions are well-formed and have the specified semantics (m denotes a value of type L).

§ 30.2.5.2

<sup>336)</sup> All implementations for which standard time units are meaningful must necessarily have a steady clock within their hardware implementation.

#### m.lock()

2 Effects: Blocks until a lock can be acquired for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

#### m.unlock()

- 3 Requires: The current execution agent shall hold a lock on m.
- 4 Effects: Releases a lock on m held by the current execution agent.

Throws: Nothing.

#### 30.2.5.3 Lockable requirements

## [thread.req.lockable.req]

A type L meets the Lockable requirements if it meets the BasicLockable requirements and the following expressions are well-formed and have the specified semantics (m denotes a value of type L).

#### m.try\_lock()

- 2 Effects: attempts to acquire a lock for the current execution agent without blocking. If an exception is thrown then a lock shall not have been acquired for the current execution agent.
- 3 Return type: bool.
- 4 Returns: true if the lock was acquired, false otherwise.

#### 30.2.5.4 TimedLockable requirements

#### [thread.req.lockable.timed]

A type L meets the TimedLockable requirements if it meets the Lockable requirements and the following expressions are well-formed and have the specified semantics (m denotes a value of type L, rel\_time denotes a value of an instantiation of duration (20.11.5), and abs\_time denotes a value of an instantiation of time\_point (20.11.6)).

#### m.try\_lock\_for(rel\_time)

- Effects: attempts to acquire a lock for the current execution agent within the relative timeout (30.2.4) specified by rel\_time. The function shall not return within the timeout specified by rel\_time unless it has obtained a lock on m for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.
- 3 Return type: bool.

2

4 Returns: true if the lock was acquired, false otherwise.

## m.try\_lock\_until(abs\_time)

- Effects: attempts to acquire a lock for the current execution agent before the absolute timeout (30.2.4) specified by abs\_time. The function shall not return before the timeout specified by abs\_time unless it has obtained a lock on m for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.
- 6 Return type: bool.
- 7 Returns: true if the lock was acquired, false otherwise.

§ 30.2.5.4

## 30.2.6 decay\_copy

# [thread.decaycopy]

In several places in this Clause the operation  $DECAY\_COPY(x)$  is used. All such uses mean call the function  $decay\_copy(x)$  and use the result, where  $decay\_copy$  is defined as follows:

```
template <class T> typename decay<T>::type decay_copy(T&& v)
  { return std::forward<T>(v); }
```

30.3 Threads [thread.threads]

30.3 describes components that can be used to create and manage threads. [Note: These threads are intended to map one-to-one with operating system threads. -end note]

## Header <thread> synopsis

```
namespace std {
  class thread;

void swap(thread& x, thread& y) noexcept;

namespace this_thread {
  thread::id get_id() noexcept;

void yield() noexcept;
  template <class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
  template <class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
  }
}
```

#### 30.3.1 Class thread

# [thread.thread.class]

The class thread provides a mechanism to create a new thread of execution, to join with a thread (i.e., wait for a thread to complete), and to perform other operations that manage and query the state of a thread. A thread object uniquely represents a particular thread of execution. That representation may be transferred to other thread objects in such a way that no two thread objects simultaneously represent the same thread of execution. A thread of execution is *detached* when no thread object represents that thread. Objects of class thread can be in a state that does not represent a thread of execution. [Note: A thread object does not represent a thread of execution after default construction, after being moved from, or after a successful call to detach or join. —end note]

```
namespace std {
  class thread {
  public:
    // types:
    class id;
    typedef implementation-defined native_handle_type; // See 30.2.3

    // construct/copy/destroy:
    thread() noexcept;
    template <class F, class ...Args> explicit thread(F&& f, Args&&... args);
    ~thread();
    thread(const thread&) = delete;
    thread(thread&&) noexcept;
    thread& operator=(const thread&) = delete;
    thread& operator=(thread&&) noexcept;

// members:
```

§ 30.3.1

```
void swap(thread&) noexcept;
      bool joinable() const noexcept;
      void join();
      void detach();
      id get_id() const noexcept;
      native_handle_type native_handle(); // See 30.2.3
      // static members:
      static unsigned hardware_concurrency() noexcept;
    };
  }
30.3.1.1 Class thread::id
                                                                                    [thread.thread.id]
  namespace std {
    class thread::id {
    public:
         id() noexcept;
    };
    bool operator==(thread::id x, thread::id y) noexcept;
    bool operator!=(thread::id x, thread::id y) noexcept;
    bool operator<(thread::id x, thread::id y) noexcept;</pre>
    bool operator<=(thread::id x, thread::id y) noexcept;</pre>
    bool operator>(thread::id x, thread::id y) noexcept;
    bool operator>=(thread::id x, thread::id y) noexcept;
    template < class charT, class traits >
      basic_ostream<charT, traits>&
        operator<< (basic_ostream<charT, traits>& out, thread::id id);
    // Hash support
    template <class T> struct hash;
    template <> struct hash<thread::id>;
An object of type thread: id provides a unique identifier for each thread of execution and a single distinct
```

- An object of type thread::id provides a unique identifier for each thread of execution and a single distinct value for all thread objects that do not represent a thread of execution (30.3.1). Each thread of execution has an associated thread::id object that is not equal to the thread::id object of any other thread of execution and that is not equal to the thread::id object of any std::thread object that does not represent threads of execution.
- 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.
- <sup>3</sup> [Note: Relational operators allow thread::id objects to be used as keys in associative containers. end note]

id() noexcept;

4

- *Effects:* Constructs an object of type id.
- 5 Postconditions: The constructed object does not represent a thread of execution.

```
bool operator==(thread::id x, thread::id y) noexcept;
```

Returns: true only if x and y represent the same thread of execution or neither x nor y represents a thread of execution.

§ 30.3.1.1

```
bool operator!=(thread::id x, thread::id y) noexcept;
7
         Returns: !(x == y)
   bool operator<(thread::id x, thread::id y) noexcept;</pre>
         Returns: A value such that operator is a total ordering as described in 25.4.
   bool operator<=(thread::id x, thread::id y) noexcept;</pre>
9
         Returns: !(y < x)
   bool operator>(thread::id x, thread::id y) noexcept;
10
         Returns: y < x
   bool operator>=(thread::id x, thread::id y) noexcept;
11
         Returns: !(x < y)
   template < class charT, class traits >
     basic_ostream<charT, traits>&
        operator<< (basic_ostream<charT, traits>&& out, thread::id id);
12
         Effects: Inserts an unspecified text representation of id into out. For two objects of type thread::id
         x and y, if x == y the thread::id objects shall have the same text representation and if x != y the
         thread::id objects shall have distinct text representations.
13
         Returns: out
   template <> struct hash<thread::id>;
14
         Requires: the template specialization shall meet the requirements of class template hash (20.8.12).
   30.3.1.2 thread constructors
                                                                                   [thread.thread.constr]
   thread() noexcept;
1
         Effects: Constructs a thread object that does not represent a thread of execution.
2
         Postcondition: get_id() == id()
   template <class F, class ...Args> explicit thread(F&& f, Args&&... args);
3
         Requires: F and each Ti in Args shall satisfy the MoveConstructible requirements. INVOKE (DECAY_-
         COPY( std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...) (20.8.2) shall be a
         valid expression.
4
         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
   § 30.3.1.2
                                                                                                      1119
```

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.

- Synchronization: The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of f.
- 6 Postconditions: get\_id() != id(). \*this represents the newly started thread.
- 7 Throws: system\_error if unable to start the new thread.
- 8 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) noexcept;

- 9 Effects: Constructs an object of type thread from x, and sets x to a default constructed state.
- Postconditions: x.get\_id() == id() and get\_id() returns the value of x.get\_id() prior to the start of construction.

#### 30.3.1.3 thread destructor

[thread.thread.destr]

~thread();

1

If joinable(), calls std::terminate(). Otherwise, has no effects. [Note: Either implicitly detaching or joining a joinable() thread in its destructor could result in difficult to debug correctness (for detach) or performance (for join) bugs encountered only when an exception is raised. Thus the programmer must ensure that the destructor is never executed while the thread is still joinable. — end note]

#### 30.3.1.4 thread assignment

[thread.thread.assign]

thread& operator=(thread&& x) noexcept;

- Effects: If joinable(), calls std::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.
- 3 Returns: \*this

## 30.3.1.5 thread members

[thread.thread.member]

```
void swap(thread& x) noexcept;
```

Effects: Swaps the state of \*this and x.

bool joinable() const noexcept;

2 Returns: get\_id() != id()

void join();

§ 30.3.1.5

```
3
         Requires: joinable() is true.
4
         Effects: Blocks until the thread represented by *this has completed.
5
         Synchronization: The completion of the thread represented by *this synchronizes with (1.10) the
         corresponding successful join() return. [Note: Operations on *this are not synchronized. — end
         note
6
         Postconditions: The thread represented by *this has completed. get id() == id().
7
         Throws: system error when an exception is required (30.2.2).
8
         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.
   void detach();
9
         Requires: joinable() is true.
10
         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.
11
         Postcondition: get_id() == id().
12
         Throws: system_error when an exception is required (30.2.2).
13
         Error conditions:
          — no_such_process — if the thread is not valid.
          — invalid_argument — if the thread is not joinable.
   id get_id() const noexcept;
14
         Returns: A default constructed id object if *this does not represent a thread, otherwise this_-
         thread::get_id() for the thread of execution represented by *this.
   30.3.1.6
             thread static members
                                                                                  [thread.thread.static]
   unsigned hardware_concurrency() noexcept;
         Returns: The number of hardware thread contexts. [Note: This value should only be considered to be
         a hint. -end note If this value is not computable or well defined an implementation should return
   30.3.1.7
             thread specialized algorithms
                                                                              [thread.thread.algorithm]
   void swap(thread& x, thread& y) noexcept;
1
         Effects: x.swap(y)
```

§ 30.3.1.7

#### 30.3.2 Namespace this\_thread

[thread.thread.this]

```
namespace std {
       namespace this_thread {
         thread::id get_id() noexcept;
         void yield() noexcept;
         template <class Clock, class Duration>
           void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
         template <class Rep, class Period>
           void sleep_for(const chrono::duration<Rep, Period>& rel_time);
       }
     }
   thread::id this_thread::get_id() noexcept;
1
        Returns: An object of type thread::id that uniquely identifies the current thread of execution. No
        other thread of execution shall have this id and this thread of execution shall always have this id. The
        object returned shall not compare equal to a default constructed thread::id.
   void this_thread::yield() noexcept;
2
        Effects: Offers the implementation the opportunity to reschedule.
3
        Synchronization: None.
   template <class Clock, class Duration>
     void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
4
        Effects: Blocks the calling thread for the absolute timeout (30.2.4) specified by abs_time.
5
        Synchronization: None.
6
         Throws: Nothing if Clock satisfies the TrivialClock requirements (20.11.3) and operations of Duration
        do not throw exceptions. [Note: instantiations of time point types and clocks supplied by the imple-
        mentation as specified in 20.11.7 do not throw exceptions. — end note]
   template <class Rep, class Period>
     void sleep_for(const chrono::duration<Rep, Period>& rel_time);
7
        Effects: Blocks the calling thread for the relative timeout (30.2.4) specified by rel_time.
8
        Synchronization: None.
9
         Throws: Nothing if operations of chrono::duration<Rep, Period> do not throw exceptions. [Note:
        instantiations of duration types supplied by the implementation as specified in 20.11.7 do not throw
        exceptions. — end note
   30.4
```

## Mutual exclusion

[thread.mutex]

This section provides mechanisms for mutual exclusion: mutexes, locks, and call once. These mechanisms ease the production of race-free programs (1.10).

Header <mutex> synopsis

§ 30.4 1122

```
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) noexcept;
  template <class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
  template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);
  struct once_flag {
    constexpr once_flag() noexcept;
    once_flag(const once_flag&) = delete;
    once_flag& operator=(const once_flag&) = delete;
  };
 template < class Callable, class ... Args>
    void call_once(once_flag& flag, Callable func, Args&&... args);
```

## 30.4.1 Mutex requirements

## [thread.mutex.requirements]

#### **30.4.1.1** In general

#### [thread.mutex.requirements.general]

A mutex object facilitates protection against data races and allows safe synchronization of data between execution agents (30.2.5). An execution agent owns a mutex from the time it successfully calls one of the lock functions until it calls unlock. Mutexes can be either recursive or non-recursive, and can grant simultaneous ownership to one or many execution agents. 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.

#### **30.4.1.2** Mutex types

#### [thread.mutex.requirements.mutex]

- The mutex types are the standard library types std::mutex, std::recursive\_mutex, std::timed\_mutex, and std::recursive\_timed\_mutex. They shall meet the requirements set out in this section. In this description, m denotes an object of a mutex type.
- The mutex types shall meet the Lockable requirements (30.2.5.3).
- The mutex types shall be DefaultConstructible and Destructible. If initialization of an object of a mutex type fails, an exception of type system\_error shall be thrown. The mutex types shall not be copyable or movable.
- <sup>4</sup> The error conditions for error codes, if any, reported by member functions of the mutex types shall be:
  - resource\_unavailable\_try\_again if any native handle type manipulated is not available.
  - operation\_not\_permitted if the thread does not have the privilege to perform the operation.

§ 30.4.1.2

- 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. For purposes of determining the existence of a data race, these behave as atomic operations (1.10). The lock and unlock operations on a single mutex shall appear to occur in a single total order. [Note: this can be viewed as the modification order (1.10) of the mutex. end note] [Note: Construction and destruction of an object of a mutex type need not be thread-safe; other synchronization should be used to ensure that mutex objects are initialized and visible to other threads. end note]
- <sup>6</sup> The expression m.lock() shall be well-formed and have the following semantics:
- Requires: If m is of type std::mutex or std::timed\_mutex, the calling thread does not own the mutex.
- 8 Effects: Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.
- 9 Postcondition: The calling thread owns the mutex.
- 10 Return type: void
- Synchronization: Prior unlock() operations on the same object shall synchronize with (1.10) this operation.
- 12 Throws: system\_error when an exception is required (30.2.2).
- 13 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.
- 14 The expression m.try\_lock() shall be well-formed and have the following semantics:
- Requires: If m is of type std::mutex or std::timed\_mutex, the calling thread does not own the mutex.
- 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 and exchange (Clause 29). —end note] An implementation should ensure that try\_lock() does not consistently return false in the absence of contending mutex acquisitions.
- 17 Return type: bool
- 18 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]
- 20 Throws: Nothing.
- The expression m.unlock() shall be well-formed and have the following semantics:
- 22 Requires: The calling thread shall own the mutex.
- 23 Effects: Releases the calling thread's ownership of the mutex.
- 24 Return type: void
- Synchronization: This operation synchronizes with (1.10) subsequent lock operations that obtain ownership on the same object.
- 26 Throws: Nothing.

§ 30.4.1.2

#### 30.4.1.2.1 Class mutex

[thread.mutex.class]

```
namespace std {
  class mutex {
  public:
     constexpr mutex() noexcept;
     ~mutex();

     mutex(const mutex&) = delete;
     mutex& operator=(const mutex&) = delete;

     void lock();
     bool try_lock();
     void unlock();

     typedef implementation-defined native_handle_type; // See 30.2.3
     native_handle_type native_handle(); // See 30.2.3
    };
}
```

- The class mutex provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a mutex object, attempts by another thread to acquire ownership of that object will fail (for try\_lock()) or block (for lock()) until the owning thread has released ownership with a call to unlock().
- <sup>2</sup> [Note: After a thread A has called unlock(), releasing a mutex, it is possible for another thread B to lock the same mutex, observe that it is no longer in use, unlock it, and destroy it, before thread A appears to have returned from its unlock call. Implementations are required to handle such scenarios correctly, as long as thread A doesn't access the mutex after the unlock call returns. These cases typically occur when a reference-counted object contains a mutex that is used to protect the reference count. end note]
- The class mutex shall satisfy all the Mutex requirements (30.4.1). It shall be a standard-layout class (Clause 9).
- <sup>4</sup> [Note: A program may deadlock if the thread that owns a mutex object calls lock() on that object. If the implementation can detect the deadlock, a resource\_deadlock\_would\_occur error condition may be observed. end note]
- The behavior of a program is undefined if it destroys a mutex object owned by any thread or a thread terminates while owning a mutex object.

# 30.4.1.2.2 Class recursive\_mutex

[thread.mutex.recursive]

```
namespace std {
  class recursive_mutex {
  public:
    recursive_mutex();
    ~recursive_mutex();

    recursive_mutex(const recursive_mutex&) = delete;
    recursive_mutex& operator=(const recursive_mutex&) = delete;

    void lock();
    bool try_lock() noexcept;
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.2.3
    native_handle_type native_handle(); // See 30.2.3
};
}
```

§ 30.4.1.2.2

The class recursive\_mutex provides a recursive mutex with exclusive ownership semantics. If one thread owns a recursive\_mutex object, attempts by another thread to acquire ownership of that object will fail (for try\_lock()) or block (for lock()) until the first thread has completely released ownership.

- <sup>2</sup> The class recursive\_mutex shall satisfy all the Mutex requirements (30.4.1). It shall be a standard-layout class (Clause 9).
- A thread that owns a recursive\_mutex object may acquire additional levels of ownership by calling lock() or try\_lock() on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a recursive\_mutex object, additional calls to try\_lock() shall fail, and additional calls to lock() shall throw an exception of type system\_error. A thread shall call unlock() once for each level of ownership acquired by calls to lock() and try\_lock(). Only when all levels of ownership have been released may ownership be acquired by another thread.
- 4 The behavior of a program is undefined if:
  - it destroys a recursive\_mutex object owned by any thread or
  - a thread terminates while owning a recursive mutex object.

#### 30.4.1.3 Timed mutex types

## [thread.timedmutex.requirements]

- The timed mutex types are the standard library types std::timed\_mutex and std::recursive\_timed\_mutex. They shall meet the requirements set out below. In this description, m denotes an object of a mutex type, rel\_time denotes an object of an instantiation of duration (20.11.5), and abs\_time denotes an object of an instantiation of time\_point (20.11.6).
- <sup>2</sup> The timed mutex types shall meet the TimedLockable requirements (30.2.5.4).
- 3 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. If m is of type std::timed\_mutex, the calling thread does not own the mutex.
- 5 Effects: The function attempts to obtain ownership of the mutex within the relative timeout (30.2.4) specified by rel\_time. If the time specified by rel\_time is less than or equal to rel\_time.zero(), the function attempts to obtain ownership without blocking (as if by calling try\_lock()). The function shall return within the timeout specified by rel\_time only if it has obtained ownership of the mutex object. [Note: As with try\_lock(), there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. end note]
- 6 Return type: bool
- <sup>7</sup> 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.
- 9 Throws: Nothing.
- 10 The expression m.try\_lock\_until(abs\_time) shall be well-formed and have the following semantics:
- 11 Requires: If m is of type std::timed\_mutex, the calling thread does not own the mutex.
- Effects: The function attempts to obtain ownership of the mutex. If abs\_time has already passed, the function attempts to obtain ownership without blocking (as if by calling try\_lock()). The function shall return before the absolute timeout (30.2.4) specified by abs\_time only if it has obtained ownership of the mutex object. [Note: As with try\_lock(), there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. end note]
- 13 Return type: bool

§ 30.4.1.3

```
14 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.

16 Throws: Nothing.

#### 30.4.1.3.1 Class timed\_mutex

[thread.timedmutex.class]

```
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
 };
}
```

- 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).
- <sup>2</sup> The class timed\_mutex shall satisfy all of the TimedMutex requirements (30.4.1.3). It shall be a standard-layout class (Clause 9).
- <sup>3</sup> 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.

#### 30.4.1.3.2 Class recursive\_timed\_mutex

[thread.timedmutex.recursive]

```
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;
```

§ 30.4.1.3.2

```
void lock();
bool try_lock() noexcept;
template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
void unlock();

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).
- <sup>2</sup> The class recursive\_timed\_mutex shall satisfy all of the TimedMutex requirements (30.4.1.3). It shall be a standard-layout class (Clause 9).
- A thread that owns a recursive\_timed\_mutex object may acquire additional levels of ownership by calling lock(), try\_lock(), try\_lock\_for(), or try\_lock\_until() on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a recursive\_timed\_mutex object, additional calls to try\_lock(), try\_lock\_for(), or try\_lock\_until() shall fail, and additional calls to lock() shall throw an exception of type system\_error. A thread shall call unlock() once for each level of ownership acquired by calls to lock(), try\_lock(), try\_lock\_for(), and try\_lock\_until(). Only when all levels of ownership have been released may ownership of the object be acquired by another thread.
- 4 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.

30.4.2 Locks [thread.lock]

- A lock is an object that holds a reference to a lockable object and may unlock the lockable object during the lock's destruction (such as when leaving block scope). An execution agent may use a lock to aid in managing ownership of a lockable object in an exception safe manner. A lock is said to own a lockable object if it is currently managing the ownership of that lockable object for an execution agent. A lock does not manage the lifetime of the lockable object it references. [Note: Locks are intended to ease the burden of unlocking the lockable object under both normal and exceptional circumstances. —end note]
- <sup>2</sup> Some lock constructors take tag types which describe what should be done with the lockable object during the lock's construction.

§ 30.4.2

30.4.2.1 Class template lock\_guard

[thread.lock.guard]

```
namespace std {
       template <class Mutex>
       class lock_guard {
       public:
         typedef Mutex mutex_type;
         explicit lock_guard(mutex_type& m);
         lock_guard(mutex_type& m, adopt_lock_t);
         ~lock_guard();
         lock_guard(lock_guard const&) = delete;
         lock_guard& operator=(lock_guard const&) = delete;
       private:
         mutex_type& pm; // exposition only
  An object of type lock_guard controls the ownership of a lockable object within a scope. A lock_guard
   object maintains ownership of a lockable object throughout the lock_guard object's lifetime (3.8). The
   behavior of a program is undefined if the lockable object referenced by pm does not exist for the entire lifetime
   of the lock_guard object. The supplied Mutex type shall meet the BasicLockable requirements (30.2.5.2).
   explicit lock_guard(mutex_type& m);
2
        Requires: If mutex_type is not a recursive mutex, the calling thread does not own the mutex m.
3
        Effects: m.lock()
4
        Postcondition: \&pm == \&m
   lock_guard(mutex_type& m, adopt_lock_t);
5
        Requires: The calling thread owns the mutex m.
6
        Postcondition: &pm == &m
7
         Throws: Nothing.
   ~lock_guard();
        Effects: pm.unlock()
   30.4.2.2
             Class template unique_lock
                                                                                     [thread.lock.unique]
     namespace std {
       template <class Mutex>
       class unique_lock {
       public:
         typedef Mutex mutex_type;
         // 30.4.2.2.1, construct/copy/destroy:
         unique_lock() noexcept;
         explicit unique_lock(mutex_type& m);
         unique_lock(mutex_type& m, defer_lock_t) noexcept;
         unique_lock(mutex_type& m, try_to_lock_t);
   § 30.4.2.2
                                                                                                      1129
```

unique\_lock(mutex\_type& m, adopt\_lock\_t);

```
template <class Clock, class Duration>
    unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
  template <class Rep, class Period>
    unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
  ~unique_lock();
  unique lock(unique lock const&) = delete;
  unique_lock& operator=(unique_lock const&) = delete;
  unique_lock(unique_lock&& u) noexcept;
  unique_lock& operator=(unique_lock&& u) noexcept;
  // 30.4.2.2.2, locking:
  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();
  // 30.4.2.2.3, modifiers:
  void swap(unique_lock& u) noexcept;
 mutex_type *release() noexcept;
  // 30.4.2.2.4, observers:
  bool owns_lock() const noexcept;
  explicit operator bool () const noexcept;
 mutex_type* mutex() const noexcept;
private:
 mutex_type *pm; // exposition only
 bool owns;
                 // exposition only
template <class Mutex>
  void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
```

- An object of type unique\_lock controls the ownership of a lockable object within a scope. Ownership of the lockable object may be acquired at construction or after construction, and may be transferred, after acquisition, to another unique\_lock object. Objects of type unique\_lock are not copyable but are movable. The behavior of a program is undefined if the contained pointer pm is not null and the lockable object pointed to by pm does not exist for the entire remaining lifetime (3.8) of the unique\_lock object. The supplied Mutex type shall meet the BasicLockable requirements (30.2.5.2).
- [Note: unique\_lock<Mutex> meets the BasicLockable requirements. If Mutex meets the Lockable requirements (30.2.5.3), unique\_lock<Mutex> also meets the Lockable requirements; if Mutex meets the TimedLockable requirements (30.2.5.4), unique\_lock<Mutex> also meets the TimedLockable requirements.

   end note

30.4.2.2.1 unique\_lock constructors, destructor, and assignment [thread.lock.unique.cons]

```
unique_lock() noexcept;
```

§ 30.4.2.2.1

```
1
         Effects: Constructs an object of type unique_lock.
2
         Postconditions: pm == 0 and owns == false.
   explicit unique_lock(mutex_type& m);
3
         Requires: If mutex type is not a recursive mutex the calling thread does not own the mutex.
4
         Effects: Constructs an object of type unique_lock and calls m.lock().
5
         Postconditions: pm == &m and owns == true.
   unique_lock(mutex_type& m, defer_lock_t) noexcept;
6
         Effects: Constructs an object of type unique_lock.
         Postconditions: pm == &m and owns == false.
   unique_lock(mutex_type& m, try_to_lock_t);
8
         Requires: The supplied Mutex type shall meet the Lockable requirements (30.2.5.3). If mutex_type is
         not a recursive mutex the calling thread does not own the mutex.
9
         Effects: Constructs an object of type unique_lock and calls m.try_lock().
10
         Postconditions: pm == &m and owns == res, where res is the value returned by the call to m.try_-
         lock().
   unique_lock(mutex_type& m, adopt_lock_t);
11
         Requires: The calling thread own the mutex.
12
         Effects: Constructs an object of type unique_lock.
13
         Postconditions: pm == &m and owns == true.
14
         Throws: Nothing.
   template <class Clock, class Duration>
      unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
15
         Requires: If mutex_type is not a recursive mutex the calling thread does not own the mutex. The
         supplied Mutex type shall meet the TimedLockable requirements (30.2.5.4).
16
         Effects: Constructs an object of type unique_lock and calls m.try_lock_until(abs_time).
17
         Postconditions: pm == &m and owns == res, where res is the value returned by the call to m.try_-
         lock_until(abs_time).
   template <class Rep, class Period>
      unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
18
         Requires: If mutex_type is not a recursive mutex the calling thread does not own the mutex. The
         supplied Mutex type shall meet the TimedLockable requirements (30.2.5.4).
19
         Effects: Constructs an object of type unique_lock and calls m.try_lock_for(rel_time).
20
         Postconditions: pm == &m and owns == res, where res is the value returned by the call to m.try_-
         lock_for(rel_time).
```

1131

§ 30.4.2.2.1

```
unique_lock(unique_lock&& u) noexcept;
21
         Postconditions: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this
         construction), u.pm == 0 and u.owns == false.
   unique_lock& operator=(unique_lock&& u) noexcept;
22
         Effects: If owns calls pm->unlock().
23
         Postconditions: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this
         construction), u.pm == 0 and u.owns == false.
24
         Note: With a recursive mutex it is possible for both *this and u to own the same mutex before the
         assignment. In this case, *this will own the mutex after the assignment and u will not. —end note]
    ~unique_lock();
25
         Effects: If owns calls pm->unlock().
   30.4.2.2.2 unique_lock locking
                                                                           [thread.lock.unique.locking]
   void lock();
1
         Effects: pm->lock()
2
         Postcondition: owns == true
3
         Throws: Any exception thrown by pm->lock(). system_error if an exception is required (30.2.2).
         system_error with an error condition of operation_not_permitted if pm is 0. system_error with
         an error condition of resource_deadlock_would_occur if on entry owns is true.
   bool try_lock();
4
         Requires: The supplied Mutex shall meet the Lockable requirements (30.2.5.3).
5
         Effects: pm->try_lock()
6
         Returns: The value returned by the call to try_lock().
7
         Postcondition: owns == res, where res is the value returned by the call to try_lock().
8
         Throws: Any exception thrown by pm->try_lock(). system_error if an exception is required (30.2.2).
         system_error with an error condition of operation_not_permitted if pm is 0. system_error with
         an error condition of resource_deadlock_would_occur if on entry owns is true.
   template <class Clock, class Duration>
     bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
9
         Requires: The supplied Mutex type shall meet the TimedLockable requirements (30.2.5.4).
10
         Effects: pm->try_lock_until(abs_time)
11
         Returns: The value returned by the call to try_lock_until(abs_time).
12
         Postcondition: owns == res, where res is the value returned by the call to try_lock_until(abs_-
         time).
13
         Throws: Any exception thrown by pm->try_lock_until(). system_error if an exception is re-
         quired (30.2.2). system_error with an error condition of operation_not_permitted if pm is 0.
         system_error with an error condition of resource_deadlock_would_occur if on entry owns is true.
```

1132

§ 30.4.2.2.2

```
template <class Rep, class Period>
     bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
         Requires: The supplied Mutex type shall meet the TimedLockable requirements (30.2.5.4).
14
15
         Effects: pm->try_lock_for(rel_time).
16
         Returns: The value returned by the call to try_lock_until(rel_time).
17
         Postcondition: owns == res, where res is the value returned by the call to try_lock_for(rel_time).
18
         Throws: Any exception thrown by pm->try_lock_for(). system_error if an exception is required
         (30.2.2). system_error with an error condition of operation_not_permitted if pm is 0. system_-
         error with an error condition of resource_deadlock_would_occur if on entry owns is true.
   void unlock();
19
         Effects: pm->unlock()
20
         Postcondition: owns == false
21
         Throws: system_error when an exception is required (30.2.2).
22
         Error conditions:
          — operation_not_permitted — if on entry owns is false.
   30.4.2.2.3 unique_lock modifiers
                                                                              [thread.lock.unique.mod]
   void swap(unique_lock& u) noexcept;
1
         Effects: Swaps the data members of *this and u.
   mutex_type *release() noexcept;
2
         Returns: The previous value of pm.
3
         Postconditions: pm == 0 and owns == false.
   template <class Mutex>
     void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
         Effects: x.swap(y)
                                                                               [thread.lock.unique.obs]
   30.4.2.2.4 unique_lock observers
   bool owns_lock() const noexcept;
         Returns: owns
   explicit operator bool() const noexcept;
         Returns: owns
   mutex_type *mutex() const noexcept;
         Returns: pm
   § 30.4.2.2.4
                                                                                                     1133
```

## 30.4.3 Generic locking algorithms

[thread.lock.algorithm]

template <class L1, class L2, class... L3> int try\_lock(L1&, L2&, L3&...);

Requires: Each template parameter type shall meet the Lockable requirements. [Note: The unique\_-lock class template meets these requirements when suitably instantiated. — 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().
- Returns: -1 if all calls to try\_lock() returned true, otherwise a 0-based index value that indicates the argument for which try\_lock() returned false.

template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);

- Requires: Each template parameter type shall meet the Lockable requirements, [Note: The unique\_-lock class template meets these requirements when suitably instantiated. end note]
- Effects: All arguments are locked via a sequence of calls to lock(), try\_lock(), or unlock() on each argument. The sequence of calls shall not result in deadlock, but is otherwise unspecified. [Note: A deadlock avoidance algorithm such as try-and-back-off must be used, but the specific algorithm is not specified to avoid over-constraining implementations. —end note] If a call to lock() or try\_lock() throws an exception, unlock() shall be called for any argument that had been locked by a call to lock() or try\_lock().

30.4.4 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.

#### 30.4.4.1 Struct once\_flag

1

1

[thread.once.onceflag]

constexpr once\_flag() noexcept;

- Effects: Constructs an object of type once\_flag.
- 2 Synchronization: The construction of a once\_flag object is not synchronized.
- Postcondition: The object's internal state is set to indicate to an invocation of call\_once with the object as its initial argument that no function has been called.

## 30.4.4.2 Function call\_once

[thread.once.callonce]

```
template<class Callable, class ...Args> void call_once(once_flag& flag, Callable&& func, Args&&... args);
```

- 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.
- Effects: An execution of call\_once that does not call its func is a passive execution. An execution of call\_once that calls its func is an active execution. An active execution shall call INVOKE (DECAY\_-COPY(std::forward<Callable>(func)), DECAY\_COPY(std::forward<Args>(args))...). If such a call to func throws an exception the execution is exceptional, otherwise it is returning. An exceptional execution shall propagate the exception to the caller of call\_once. Among all executions of call\_once for any given once\_flag: at most one shall be a returning execution; if there is a returning execution, it shall be the last active execution; and there are passive executions only if there is a

§ 30.4.4.2

returning execution. [Note: passive executions allow other threads to reliably observe the results produced by the earlier returning execution.  $-end\ note$ ]

- 3 Synchronization: For any given once\_flag: all active executions occur in a total order; completion of an active execution synchronizes with (1.10) the start of the next one in this total order; and the returning execution synchronizes with the return from all passive executions.
- 4 Throws: 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.
- 6 [Example:

```
// 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, &information::verifier, *this); }
— end example]
```

## 30.5 Condition variables

# [thread.condition]

- 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

§ 30.5

- 3. the reacquisition of the lock.
- <sup>4</sup> 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.
- <sup>5</sup> Condition variable construction and destruction need not be synchronized.

## Header condition\_variable synopsis

```
namespace std {
   class condition_variable;
   class condition_variable_any;

   void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);
   enum class cv_status { no_timeout, timeout };
}

void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);
```

- 6 Requires: 1k 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.
- 7 Effects: transfers ownership of the lock associated with 1k 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

```
lk.unlock();
cond.notify_all();
```

- Synchronization: The call to notify\_all\_at\_thread\_exit and the completion of the destructors for all the current thread's variables of thread storage duration synchronize with (1.10) calls to functions waiting on cond.
- 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

[thread.condition.condvar]

```
namespace std {
  class condition_variable {
   public:
      condition_variable();
      ~condition_variable();
      condition_variable(const condition_variable&) = delete;
```

§ 30.5.1

```
condition_variable& operator=(const condition_variable&) = delete;
         void notify_one() noexcept;
         void notify_all() noexcept;
         void wait(unique_lock<mutex>& lock);
         template <class Predicate>
           void wait(unique_lock<mutex>& lock, Predicate pred);
         template <class Clock, class Duration>
           cv_status wait_until(unique_lock<mutex>& lock,
                                 const chrono::time_point<Clock, Duration>& abs_time);
         template <class Clock, class Duration, class Predicate>
           bool wait_until(unique_lock<mutex>& lock,
                            const chrono::time_point<Clock, Duration>& abs_time,
                            Predicate pred);
         template <class Rep, class Period>
           cv_status wait_for(unique_lock<mutex>& lock,
                               const chrono::duration<Rep, Period>& rel_time);
         template <class Rep, class Period, class Predicate>
           bool wait_for(unique_lock<mutex>& lock,
                          const chrono::duration<Rep, Period>& rel_time,
                         Predicate pred);
         typedef implementation-defined native_handle_type; // See 30.2.3
                                                              // See 30.2.3
         native_handle_type native_handle();
     }
  The class condition variable shall be a standard-layout class (Clause 9).
   condition_variable();
2
        Effects: Constructs an object of type condition_variable.
3
        Throws: system_error when an exception is required (30.2.2).
4
        Error conditions:
          — resource_unavailable_try_again — if some non-memory resource limitation prevents initial-
             ization.
   ~condition_variable();
5
        Requires: There shall be no thread blocked on *this. [Note: That is, all threads shall have been
        notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules,
        which would have required all wait calls to happen before destruction. Only the notification to unblock
        the wait must happen before destruction. The user must take care to ensure that no threads wait on
        *this once the destructor has been started, especially when the waiting threads are calling the wait
        functions in a loop or using the overloads of wait, wait_for, or wait_until that take a predicate.
           end \ note
6
        Effects: Destroys the object.
   void notify_one() noexcept;
        Effects: If any threads are blocked waiting for *this, unblocks one of those threads.
```

§ 30.5.1

void notify\_all() noexcept;

```
Effects: Unblocks all threads that are blocked waiting for *this.
    void wait(unique_lock<mutex>& lock);
9
         Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
           — no other thread is waiting on this condition_variable object or
             lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently
              waiting (via wait or timed_wait) threads.
10
         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
11
         Postcondition: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.
12
         Throws: system error when an exception is required (30.2.2).
13
         Error conditions:
           — equivalent error condition from lock.lock() or lock.unlock().
    template <class Predicate>
      void wait(unique_lock<mutex>& lock, Predicate pred);
         Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
           — 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.
15
         Effects:
           while (!pred())
             wait(lock);
16
         Postcondition: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.
17
         Throws: std::system error when an exception is required (30.2.2).
18
         Error conditions:
           — equivalent error condition from lock.lock() or lock.unlock().
    template <class Clock, class Duration>
      cv_status wait_until(unique_lock<mutex>& lock,
                            const chrono::time_point<Clock, Duration>& abs_time);
    § 30.5.1
                                                                                                       1138
```

19 Requires: lock.owns\_lock() is true and lock.mutex() is locked by the calling thread, and either

- no other thread is waiting on this condition\_variable object or
- lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait\_for, or wait\_until) threads.
- 20 Effects:
  - Atomically calls lock.unlock() and blocks on \*this.
  - When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.
  - The function will unblock when signaled by a call to notify\_one(), a call to notify\_all(), expiration of the absolute timeout (30.2.4) specified by abs\_time, or spuriously.
  - If the function exits via an exception, lock.lock() shall be called prior to exiting the function scope.
- 21 Postcondition: lock.owns\_lock() is true and lock.mutex() is locked by the calling thread.
- Returns: cv\_status::timeout if the absolute timeout (30.2.4) specified by abs\_time expired, otherwise cv\_status::no\_timeout.
- 23 Throws: system\_error when an exception is required (30.2.2).
- 24 Error conditions:
  - equivalent error condition from lock.lock() or lock.unlock().

- 25 Requires: lock.owns\_lock() is true and lock.mutex() is locked by the calling thread, and either
  - no other thread is waiting on this condition\_variable object or
  - lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait\_for, or wait\_until) threads.
- 26 Effects: as if

```
return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

- Returns: cv\_status::timeout if the relative timeout (30.2.4) specified by rel\_time expired, otherwise cv\_status::no\_timeout.
- Postcondition: lock.owns\_lock() is true and lock.mutex() is locked by the calling thread.
- 29 Throws: system error when an exception is required (30.2.2).
- 30 Error conditions:
  - equivalent error condition from lock.lock() or lock.unlock().

Requires: lock.owns\_lock() is true and lock.mutex() is locked by the calling thread, and either

— no other thread is waiting on this condition\_variable object or

```
lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently
              waiting (via wait or timed_wait) threads.
32
         Effects:
           while (!pred())
             if (wait_until(lock, abs_time) == cv_status::timeout)
               return pred();
           return true;
33
         Returns: pred()
34
         Postcondition: lock.owns lock() is true and lock.mutex() is locked by the calling thread.
35
         [Note: The returned value indicates whether the predicate evaluated to true regardless of whether
         the timeout was triggered. — end note
36
         Throws: std::system_error when an exception is required (30.2.2).
37
         Error conditions:
           — equivalent error condition from lock.lock() or lock.unlock().
    template <class Rep, class Period, class Predicate>
      bool wait_for(unique_lock<mutex>& lock,
                    const chrono::duration<Rep, Period>& rel_time,
                    Predicate pred);
38
         Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
           — no other thread is waiting on this condition_variable object or
           — lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently
              waiting (via wait, wait_for, or wait_until) threads.
39
         Effects: as if
           return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));
40
         [Note: There is no blocking if pred() is initially true, even if the timeout has already expired. — end
41
         Postcondition: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.
42
         Returns: pred()
43
         Note: The returned value indicates whether the predicate evaluates to true regardless of whether the
         timeout was triggered. — end note]
44
         Throws: system_error when an exception is required (30.2.2).
45
         Error conditions:
           — equivalent error condition from lock.lock() or lock.unlock().
```

## 30.5.2 Class condition\_variable\_any

4

5

## [thread.condition.condvarany]

A Lock type shall meet the BasicLockable requirements (30.2.5.2). [Note: All of the standard mutex types meet this requirement. If a Lock type other than one of the standard mutex types or a unique\_lock wrapper for a standard mutex type is used with condition\_variable\_any, the user must ensure that any necessary synchronization is in place with respect to the predicate associated with the condition\_variable\_any instance. -end note

```
namespace std {
       class condition_variable_any {
       public:
         condition_variable_any();
         ~condition_variable_any();
         condition_variable_any(const condition_variable_any&) = delete;
         condition_variable_any& operator=(const condition_variable_any&) = delete;
         void notify_one() noexcept;
         void notify_all() noexcept;
         template <class Lock>
           void wait(Lock& lock);
         template <class Lock, class Predicate>
           void wait(Lock& lock, Predicate pred);
         template <class Lock, class Clock, class Duration>
           cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
         template <class Lock, class Clock, class Duration, class Predicate>
           bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time,
             Predicate pred);
         template <class Lock, class Rep, class Period>
           cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
         template <class Lock, class Rep, class Period, class Predicate>
           bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time,
             Predicate pred);
       };
     }
   condition_variable_any();
2
        Effects: Constructs an object of type condition variable any.
3
        Throws: bad_alloc or system_error when an exception is required (30.2.2).
        Error conditions:
          — 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.
   ~condition_variable_any();
```

Requires: There shall be no thread blocked on \*this. [Note: That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait must happen before destruction. The user must take care to ensure that no threads wait on \*this once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of wait, wait\_for, or wait\_until that take a predicate. -end note

```
6
         Effects: Destroys the object.
    void notify_one() noexcept;
         Effects: If any threads are blocked waiting for *this, unblocks one of those threads.
    void notify_all() noexcept;
         Effects: Unblocks all threads that are blocked waiting for *this.
    template <class Lock>
      void wait(Lock& lock);
         Note: if any of the wait functions exits via an exception, it is unspecified whether the Lock is held.
         One can use a Lock type that allows to query that, such as the unique_lock wrapper.
10
         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
11
         Postcondition: lock is locked by the calling thread.
12
         Throws: system_error when an exception is required (30.2.2).
13
         Error conditions:
           — equivalent error condition from lock.lock() or lock.unlock().
    template <class Lock, class Predicate>
      void wait(Lock& lock, Predicate pred);
14
         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);
15
         Effects:
           — Atomically calls lock.unlock() and blocks on *this.
           — When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.
           — The function will unblock when signaled by a call to notify_one(), a call to notify_all(),
              expiration of the absolute timeout (30.2.4) specified by abs_time, or spuriously.
```

```
— If the function exits via an exception, lock.lock() shall be called prior to exiting the function
              scope.
16
         Postcondition: lock is locked by the calling thread.
17
         Returns: cv status::timeout if the absolute timeout (30.2.4) specified by abs time expired, other-
         wise cv_status::no_timeout.
18
         Throws: system_error when an exception is required (30.2.2).
19
         Error conditions:
           — equivalent error condition from lock.lock() or lock.unlock().
    template <class Lock, class Rep, class Period>
      cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
20
         Effects: as if
           return wait_until(lock, chrono::steady_clock::now() + rel_time);
21
         Returns: cv_status::timeout if the relative timeout (30.2.4) specified by rel_time expired, otherwise
         cv_status::no_timeout.
^{22}
         Postcondition: lock is locked by the calling thread.
23
         Throws: system_error when an exception is required (30.2.2).
24
         Error conditions:
           — equivalent error condition from lock.lock() or lock.unlock().
    template <class Lock, class Clock, class Duration, class Predicate>
      bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
25
         Effects:
           while (!pred())
             if (wait_until(lock, abs_time) == cv_status::timeout)
               return pred();
           return true;
26
         Returns: pred()
27
         Note: The returned value indicates whether the predicate evaluates to true regardless of whether the
         timeout was triggered. — end note]
    template <class Lock, class Rep, class Period, class Predicate>
      bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);
28
         Effects: as if
           return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));
```

```
[Note: There is no blocking if pred() is initially true, even if the timeout has already expired. — end note]
```

- Postcondition: lock is locked by the calling thread.
- 31 Returns: pred()
- [ Note: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered. end note]
- 33 Throws: system error when an exception is required (30.2.2).
- 34 Error conditions:
  - equivalent error condition from lock.lock() or lock.unlock().

# 30.6 Futures [futures]

### 30.6.1 Overview

[futures.overview]

30.6 describes components that a C++ program can use to retrieve in one thread the result (value or exception) from a function that has run in the same thread or another thread. [Note: These components are not restricted to multi-threaded programs but can be useful in single-threaded programs as well. — end note]

```
Header <future> synopsis
```

```
namespace std {
  enum class future_errc {
   broken_promise,
    future_already_retrieved,
    promise_already_satisfied,
    no_state
  };
  enum class launch : unspecified {
    async = unspecified,
    deferred = unspecified,
    imple mentation-defined
  };
  enum class future_status {
    ready,
    timeout,
    deferred
  };
  template <> struct is_error_code_enum<future_errc> : public true_type { };
  error_code make_error_code(future_errc e) noexcept;
  error_condition make_error_condition(future_errc e) noexcept;
  const error_category& future_category() noexcept;
  class future_error;
  template <class R> class promise;
  template <class R> class promise<R&>;
  template <> class promise<void>;
  template <class R>
```

```
void swap(promise<R>& x, promise<R>& y) noexcept;
       template <class R, class Alloc>
         struct uses_allocatororomise<R>, Alloc>;
       template <class R> class future;
       template <class R> class future<R&>;
      template <> class future<void>;
       template <class R> class shared_future;
       template <class R> class shared_future<R&>;
       template <> class shared_future<void>;
       template <class> class packaged_task;
       template <class R, class... ArgTypes>
         class packaged_task<R(ArgTypes...)>;
       template <class R>
         void swap(packaged_task<R(ArgTypes...)>&, packaged_task<R(ArgTypes...)>&) noexcept;
       template <class R, class Alloc>
         struct uses_allocator<packaged_task<R>, Alloc>;
      template <class F, class... Args>
         future<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);
  The enum type launch is an implementation-defined bitmask type (17.5.2.1.3) with launch::async and
   launch::deferred denoting individual bits. [Note: Implementations can provide bitmasks to specify re-
   strictions on task interaction by functions launched by async() applicable to a corresponding subset of
   available launch policies. Implementations can extend the behavior of the first overload of async() by
   adding their extensions to the launch policy under the "as if" rule. — end note]
   30.6.2 Error handling
                                                                                      [futures.errors]
   const error_category& future_category() noexcept;
1
        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".
   error_code make_error_code(future_errc e) noexcept;
3
        Returns: error_code(static_cast<int>(e), future_category()).
   error_condition make_error_condition(future_errc e) noexcept;
4
        Returns: error_condition(static_cast<int>(e), future_category()).
   § 30.6.2
                                                                                                   1145
```

## 30.6.3 Class future error

[futures.future error]

```
namespace std {
  class future_error : public logic_error {
  public:
     future_error(error_code ec); // exposition only

     const error_code& code() const noexcept;
     const char* what() const noexcept;
  };
}

const error_code& code() const noexcept;
  Returns: The value of ec that was passed to the object's constructor.

const char *what() const noexcept;
```

2 Returns: An NTBS incorporating code().message().

#### 30.6.4 Shared state

1

[futures.state]

- Many of the classes introduced in this sub-clause use some state to communicate results. This *shared state* consists of some state information and some (possibly not yet evaluated) *result*, which can be a (possibly void) value or an exception. [*Note:* Futures, promises, and tasks defined in this clause reference such shared state. *end note*]
- [ Note: The result can be any kind of object including a function to compute that result, as used by async when policy is launch::deferred. end note]
- <sup>3</sup> An asynchronous return object is an object that reads results from an shared state. A waiting function of an asynchronous return object is one that potentially blocks to wait for the shared state to be made ready. If a waiting function can return before the state is made ready because of a timeout (30.2.5), then it is a timed waiting function, otherwise it is a non-timed waiting function.
- <sup>4</sup> An asynchronous provider is an object that provides a result to a shared state. The result of a shared state is set by respective functions on the asynchronous provider. [Note: Such as promises or tasks. end note] The means of setting the result of a shared state is specified in the description of those classes and functions that create such a state object.
- When an asynchronous return object or an asynchronous provider is said to release its shared state, it means:
  - if the return object or provider holds the last reference to its shared state, the shared state is destroyed; and
  - the return object or provider gives up its reference to its shared state.
- <sup>6</sup> When an asynchronous provider is said to make its shared state ready, it means:
  - first, the provider marks its shared state as ready; and
  - second, the provider unblocks any execution agents waiting for its shared state to become ready.
- When an asynchronous provider is said to abandon its shared state, it means:
  - first, if that state is not ready, the provider
    - stores an exception object of type future\_error with an error condition of broken\_promise within its shared state; and then

- makes its shared state ready;
- second, the provider releases its shared state.
- A shared state is *ready* only if it holds a value or an exception ready for retrieval. Waiting for a shared state to become ready may invoke code to compute the result on the waiting thread if so specified in the description of the class or function that creates the state object.
- <sup>9</sup> Calls to functions that successfully set the stored result of a shared state synchronize with (1.10) calls to functions successfully detecting the ready state resulting from that setting. The storage of the result (whether normal or exceptional) into the shared state synchronizes with (1.10) the successful return from a call to a waiting function on the shared state.
- Some functions (e.g., promise::set\_value\_at\_thread\_exit) delay making the shared state ready until the calling thread exits. The destruction of each of that thread's objects with thread storage duration (3.7.2) is sequenced before making that shared state ready.
- Access to the result of the same shared state may conflict (1.10). [Note: this explicitly specifies that the result of the shared state is visible in the objects that reference this state in the sense of data race avoidance (17.6.5.9). For example, concurrent accesses through references returned by shared\_future::get() (30.6.7) must either use read-only operations or provide additional synchronization. —end note]

## 30.6.5 Class template promise

[futures.promise]

```
namespace std {
  template <class R>
  class promise {
 public:
    promise();
    template <class Allocator>
      promise(allocator_arg_t, const Allocator& a);
    promise(promise&& rhs) noexcept;
    promise(const promise& rhs) = delete;
    ~promise();
    // assignment
    promise& operator=(promise&& rhs) noexcept;
    promise& operator=(const promise& rhs) = delete;
    void swap(promise& other) noexcept;
    // retrieving the result
    future<R> get_future();
    // setting the result
    void set_value(see below);
    void set_exception(exception_ptr p);
    // setting the result with deferred notification
    void set_value_at_thread_exit(const R& r);
    void set_value_at_thread_exit(see below);
    void set_exception_at_thread_exit(exception_ptr p);
  template <class R>
    void swap(promise<R>& x, promise<R>& y) noexcept;
  template <class R, class Alloc>
    struct uses_allocatororomise<R>, Alloc>;
}
```

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.

<sup>2</sup> The set\_value, set\_exception, set\_value\_at\_thread\_exit, and set\_exception\_at\_thread\_exit member functions behave as though they acquire a single mutex associated with the promise object while updating the promise object.

```
template <class R, class Alloc>
      struct uses_allocator<promise<R>, Alloc>
        : true_type { };
3
         Requires: Alloc shall be an Allocator (17.6.3.5).
    promise();
    template <class Allocator>
      promise(allocator_arg_t, const Allocator& a);
         Effects: constructs a promise object and a shared state. The second constructor uses the allocator a
         to allocate memory for the shared state.
    promise(promise&& rhs) noexcept;
5
         Effects: constructs a new promise object and transfers ownership of the shared state of rhs (if any)
         to the newly-constructed object.
6
         Postcondition: rhs has no shared state.
    ~promise();
         Effects: Abandons any shared state (30.6.4).
    promise& operator=(promise&& rhs) noexcept;
8
         Effects: Abandons any shared state (30.6.4) and then as if promise(std::move(rhs)).swap(*this).
9
         Returns: *this.
    void swap(promise& other) noexcept;
10
         Effects: Exchanges the shared state of *this and other.
11
         Postcondition: *this has the shared state (if any) that other had prior to the call to swap. other
         has the shared state (if any) that *this had prior to the call to swap.
    future<R> get_future();
12
         Returns: A future < R > object with the same shared state as *this.
13
         Throws: future_error if *this has no shared state or if get_future has already been called on a
         promise with the same shared state as *this.
```

§ 30.6.5

future\_already\_retrieved if get\_future has already been called on a promise with the same

14

Error conditions:

shared state as \*this.

15

19

20

21

22

24

25

```
— no_state if *this has no shared state.
    void promise::set_value(const R& r);
    void promise::set_value(R&& r);
    void promise<R&>::set_value(R& r);
    void promise<void>::set_value();
          Effects: atomically stores the value \mathbf{r} in the shared state and makes that state ready (30.6.4).
          Throws:
16
           — future_error if its shared 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.
17
         Error conditions:
           — promise_already_satisfied if its shared state already has a stored value or exception.
           — no_state if *this has no shared state.
    void set_exception(exception_ptr p);
18
          Effects: atomically stores the exception pointer p in the shared state and makes that state ready (30.6.4).
          Throws: future error if its shared state already has a stored value or exception.
          Error conditions:
           — promise_already_satisfied if its shared state already has a stored value or exception.
           — no_state if *this has no shared state.
    void promise::set_value_at_thread_exit(const R& r);
    void promise::set_value_at_thread_exit(R&& r);
    void promise<R&>::set_value_at_thread_exit(R& r);
    void promise<void>::set_value_at_thread_exit();
          Effects: Stores the value r in the shared state without making that state ready immediately. Schedules
         that state to be made ready when the current thread exits, after all objects of thread storage duration
         associated with the current thread have been destroyed.
          Throws: future_error if an error condition occurs.
23
          Error conditions:
           — promise already satisfied if its shared state already has a stored value or exception.
           — no_state if *this has no shared state.
    void promise::set_exception_at_thread_exit(exception_ptr p);
         Effects: Stores the exception pointer p in the shared state without making that state ready immediately.
         Schedules that state to be made ready when the current thread exits, after all objects of thread storage
         duration associated with the current thread have been destroyed.
          Throws: future_error if an error condition occurs.
26
          Error conditions:
```

```
— promise_already_satisfied if its shared state already has a stored value or exception.
```

— no\_state if \*this has no shared state.

```
template <class R>
  void swap(promise<R>& x, promise<R>& y);
  Effects: x.swap(y).
```

## 30.6.6 Class template future

27

## [futures.unique\_future]

- The class template future defines a type for asynchronous return objects which do not share their shared state with other asynchronous return objects. A default-constructed future object has no shared state. A future object with shared state can be created by functions on asynchronous providers (30.6.4) or by the move constructor and shares its shared state with the original asynchronous provider. The result (value or exception) of a future object can be set by calling a respective function on an object that shares the same shared state.
- <sup>2</sup> [Note: Member functions of future do not synchronize with themselves or with member functions of shared\_future. end note]
- The effect of calling any member function other than the destructor, the move-assignment operator, or valid on a future object for which valid() == false is undefined. [Note: Implementations are encouraged to detect this case and throw an object of type future\_error with an error condition of future\_errc::no\_-state. end note]

```
namespace std {
  template <class R>
 class future {
  public:
    future() noexcept;
    future(future &&) noexcept;
    future(const future& rhs) = delete;
    ~future():
    future& operator=(const future& rhs) = delete;
    future& operator=(future&&) noexcept;
    shared_future<R> share();
    // retrieving the value
    see below get();
    // functions to check state
    bool valid() const noexcept;
    void wait() const;
    template <class Rep, class Period>
      future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
    template <class Clock, class Duration>
      future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
  };
```

The implementation shall provide the template future and two specializations, future<R&> and future< void>. These differ only in the return type and return value of the member function get, as set out in its description, below.

```
future() noexcept;
```

```
Effects: constructs an empty future object that does not refer to an shared state.
6
         Postcondition: valid() == false.
    future(future&& rhs) noexcept;
7
         Effects: move constructs a future object that refers to the shared state that was originally referred
         to by rhs (if any).
8
         Postconditions:
           — valid() returns the same value as rhs.valid() prior to the constructor invocation.
           - rhs.valid() == false.
    ~future();
9
         Effects:
           — releases any shared state (30.6.4);
           destroys *this.
    future& operator=(future&& rhs) noexcept;
10
         Effects:
           — releases any shared state (30.6.4).
           — move assigns the contents of rhs to *this.
11
         Postconditions:
           — valid() returns the same value as rhs.valid() prior to the assignment.
           — rhs.valid() == false.
    shared_future<R> share();
12
         Returns: shared_future<R>(std::move(*this)).
13
         Postcondition: valid() == false.
   R future::get();
   R& future<R&>::get();
    void future<void>::get();
14
         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.
15
         Effects: wait()s until the shared state is ready, then retrieves the value stored in the shared state.
16
         Returns:
           — future::get() returns the value stored in the object's shared 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 shared state.
    § 30.6.6
                                                                                                        1151
```

```
— future<void>::get() returns nothing.
17
          Throws: the stored exception, if an exception was stored in the shared state.
18
          Postcondition: valid() == false.
    bool valid() const noexcept;
19
         Returns: true only if *this refers to a shared state.
    void wait() const;
20
         Effects: blocks until the shared state is ready.
    template <class Rep, class Period>
      future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
21
          Effects: none if the shared state contains a deferred function (30.6.8), otherwise blocks until the shared
         state is ready or until the relative timeout (30.2.4) specified by rel_time has expired.
22
          Returns:
           — future status::deferred if the shared state contains a deferred function.
           — future status::ready if the shared state is ready.
           — future_status::timeout if the function is returning because the relative timeout (30.2.4) spec-
              ified by rel time has expired.
    template <class Clock, class Duration>
      future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
23
          Effects: none if the shared state contains a deferred function (30.6.8), otherwise blocks until the shared
         state is ready or until the absolute timeout (30.2.4) specified by abs time has expired.
24
          Returns:
           — future_status::deferred if the shared state contains a deferred function.
           — future_status::ready if the shared state is ready.
           — future_status::timeout if the function is returning because the absolute timeout (30.2.4) spec-
              ified by abs_time has expired.
```

### 30.6.7 Class template shared future

### [futures.shared future]

- The class template shared\_future defines a type for asynchronous return objects which may share their shared state with other asynchronous return objects. A default-constructed shared\_future object has no shared state. A shared\_future object with shared state can be created by conversion from a future object and shares its shared state with the original asynchronous provider (30.6.4) of the shared state. The result (value or exception) of a shared\_future object can be set by calling a respective function on an object that shares the same shared state.
- <sup>2</sup> [Note: Member functions of shared\_future do not synchronize with themselves, but they synchronize with the shared shared state. end note]
- <sup>3</sup> The effect of calling any member function other than the destructor, the move-assignment operator, or valid() on a shared\_future object for which valid() == false is undefined. [Note: Implementations are encouraged to detect this case and throw an object of type future\_error with an error condition of future\_erro::no\_state. end note]

```
namespace std {
    template <class R>
    class shared_future {
    public:
      shared_future() noexcept;
      shared_future(const shared_future& rhs);
      shared_future(future<R>&&) noexcept;
      shared_future(shared_future&& rhs) noexcept;
      ~shared_future();
      shared_future& operator=(const shared_future& rhs);
      shared_future& operator=(shared_future&& rhs) noexcept;
      // retrieving the value
      see below get() const;
      // functions to check state
      bool valid() const noexcept;
      void wait() const;
      template <class Rep, class Period>
        future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
      template <class Clock, class Duration>
        future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
  }
The implementation shall provide the template shared_future and two specializations, shared_future<R&>
and shared_future<void>. These differ only in the return type and return value of the member function
get, as set out in its description, below.
shared_future() noexcept;
      Effects: constructs an empty shared_future object that does not refer to an shared state.
      Postcondition: valid() == false.
shared_future(const shared_future& rhs);
      Effects: constructs a shared_future object that refers to the same shared state as rhs (if any).
      Postcondition: valid() returns the same value as rhs.valid().
shared_future(future<R>&& rhs) noexcept;
shared_future(shared_future&& rhs) noexcept;
      Effects: move constructs a shared_future object that refers to the shared state that was originally
     referred to by rhs (if any).
     Postconditions:
       — valid() returns the same value as rhs.valid() returned prior to the constructor invocation.
       — rhs.valid() == false.
~shared_future();
     Effects:
§ 30.6.7
                                                                                                   1153
```

5

6

7

8

9

10

11

```
— releases any shared state (30.6.4);
           destroys *this.
    shared_future& operator=(shared_future&& rhs) noexcept;
12
         Effects:
           — releases any shared state (30.6.4);
           — move assigns the contents of rhs to *this.
13
         Postconditions:
           — valid() returns the same value as rhs.valid() returned prior to the assignment.
           - rhs.valid() == false.
    shared_future& operator=(const shared_future& rhs);
14
         Effects:
           — releases any shared state (30.6.4);
           — assigns the contents of rhs to *this. [Note: As a result, *this refers to the same shared state
              as rhs (if any). — end note]
15
         Postconditions: valid() == rhs.valid().
    const R& shared_future::get() const;
    R& shared_future<R&>::get() const;
    void shared_future<void>::get() const;
16
         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.
17
         Note: access to a value object stored in the shared state is unsynchronized, so programmers should
         apply only those operations on R that do not introduce a data race (1.10).
18
         Effects: wait()s until the shared state is ready, then retrieves the value stored in the shared state.
19
         Returns:
              shared_future::get() returns a const reference to the value stored in the object's shared state.
              Note: Access through that reference after the shared state has been destroyed produces undefined
              behavior; this can be avoided by not storing the reference in any storage with a greater lifetime
              than the shared_future object that returned the reference. — end note]
           — shared_future<R&>::get() returns the reference stored as value in the object's shared state.
              shared_future<void>::get() returns nothing.
20
         Throws: the stored exception, if an exception was stored in the shared state.
    bool valid() const noexcept;
21
         Returns: true only if *this refers to a shared state.
```

```
void wait() const;
```

Effects: blocks until the shared state is ready.

```
template <class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
```

Effects: none if the shared state contains a deferred function (30.6.8), otherwise blocks until the shared state is ready or until the relative timeout (30.2.4) specified by rel\_time has expired.

24 Returns:

22

- future\_status::deferred if the shared state contains a deferred function.
- future\_status::ready if the shared state is ready.
- future\_status::timeout if the function is returning because the relative timeout (30.2.4) specified by rel\_time has expired.

```
template <class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
```

- Effects: none if the shared state contains a deferred function (30.6.8), otherwise blocks until the shared state is ready or until the absolute timeout (30.2.4) specified by abs\_time has expired.
- 26 Returns:
  - future status::deferred if the shared state contains a deferred function.
  - future status::ready if the shared state is ready.
  - future\_status::timeout if the function is returning because the absolute timeout (30.2.4) specified by abs\_time has expired.

# 30.6.8 Function template async

[futures.async]

The function template **async** provides a mechanism to launch a function potentially in a new thread and provides the result of the function in a **future** object with which it shares a shared state.

```
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. INVOKE (DECAY\_-COPY(std::forward<F>(f)), DECAY\_COPY(std::forward<Args>(args))...) (20.8.2, 30.3.1.2) shall be a valid expression.
- 3 Effects: The first function behaves the same as a call to the second function with a policy argument of launch::async | launch::deferred and the same arguments for F and Args. The second function creates a shared state that is associated with the returned future object. The further behavior of the second function depends on the policy argument as follows (if more than one of these conditions applies, the implementation may choose any of the corresponding policies):

— if policy & launch::async is non-zero — calls INVOKE (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 DECAY\_COPY() being evaluated in the thread that called async. Any return value is stored as the result in the shared state. Any exception propagated from the execution of INVOKE(DECAY\_COPY(std::forward<F>(f)), DECAY\_COPY(std::forward<Args>(args))...) is stored as the exceptional result in the shared state. The thread object is stored in the shared state and affects the behavior of any asynchronous return objects that reference that state.

- if policy & launch::deferred is non-zero Stores DECAY\_COPY(std::forward<F>(f)) and DECAY\_COPY(std::forward<Args>(args))... in the shared state. These copies of f and args constitute a deferred function. Invocation of the deferred function evaluates INVOKE(g, xyz) where g is the stored value of DECAY\_COPY(std::forward<F>(f)) and xyz is the stored copy of DECAY\_COPY(std::forward<Args>(args)).... The shared state is not made ready until the function has completed. The first call to a non-timed waiting function (30.6.4) on an asynchronous return object referring to this shared state shall invoke the deferred function in the thread that called the waiting function. Once evaluation of INVOKE(g, xyz) begins, the function is no longer considered deferred. [Note: If this policy is specified together with other policies, such as when using a policy value of launch::async | launch::deferred, implementations should defer invocation or the selection of the policy when no more concurrency can be effectively exploited. end note]
- 4 Returns: An object of type future<typename result\_of<F(Args...)>::type> that refers to the shared state created by this call to async.
- 5 Synchronization: Regardless of the provided policy argument,
  - the invocation of async synchronizes with (1.10) the invocation of f. [Note: This statement applies even when the corresponding future object is moved to another thread. end note]; and
  - the completion of the function f is sequenced before (1.10) the shared state is made ready. [Note: f might not be called at all, so its completion might never happen. end note]

If the implementation chooses the launch::async policy,

- a call to a waiting function on an asynchronous return object that shares the shared state created by this async call shall block until the associated thread has completed, as if joined (30.3.1.5);
- the associated thread completion synchronizes with (1.10) the return from the first function that successfully detects the ready status of the shared state or with the return from the last function that releases the shared state, whichever happens first.
- 6 Throws: system\_error if policy is launch::async and the implementation is unable to start a new thread.
- 7 Error conditions:
  - resource\_unavailable\_try\_again if policy is launch::async and the system is unable to start a new thread.
- 8 Remarks: The first signature shall not participate in overload resolution if decay<F>::type is std:: launch.
- 9 [Example:

```
int work1(int value);
int work2(int value);
int work(int value) {
  auto handle = std::async([=]{ return work2(value); });
  int tmp = work1(value);
  return tmp + handle.get();  // #1
}
```

[Note: Line #1 might not result in concurrency because the async call uses the default policy, which may use launch::deferred, in which case the lambda might not be invoked until the get() call; in that case, work1 and work2 are called on the same thread and there is no concurrency. —end note] —end example]

## 30.6.9 Class template packaged\_task

[futures.task]

- The class template packaged\_task defines a type for wrapping a function or callable object so that the return value of the function or callable object is stored in a future when it is invoked.
- When the packaged\_task object is invoked, its stored task is invoked and the result (whether normal or exceptional) stored in the shared state. Any futures that share the shared state will then be able to access the stored result.

```
namespace std {
  template<class> class packaged_task; // undefined
  template < class R, class... ArgTypes>
 class packaged_task<R(ArgTypes...)> {
  public:
    // construction and destruction
    packaged_task() noexcept;
    template <class F>
      explicit packaged_task(F&& f);
    template <class F, class Allocator>
      explicit packaged_task(allocator_arg_t, const Allocator& a, F&& f);
    ~packaged_task();
    // no copy
    packaged_task(packaged_task&) = delete;
    packaged_task& operator=(packaged_task&) = delete;
    // move support
    packaged_task(packaged_task&& rhs) noexcept;
    packaged_task& operator=(packaged_task&& rhs) noexcept;
    void swap(packaged_task& other) noexcept;
    bool valid() const noexcept;
    // result retrieval
    future<R> get_future();
    // execution
    void operator()(ArgTypes...);
    void make_ready_at_thread_exit(ArgTypes...);
    void reset();
  };
 template <class R, class... ArgTypes>
```

```
void swap(packaged_task<R(ArgTypes...)>& x, packaged_task<R(ArgTypes...)>& y) noexcept;
       template <class R, class Alloc>
          struct uses_allocator<packaged_task<R>, Alloc>;
     }
                                                                                  [futures.task.members]
   30.6.9.1
             packaged_task member functions
   packaged_task() noexcept;
         Effects: constructs a packaged_task object with no shared state and no stored task.
   template <class F>
     packaged_task(F&& f);
   template <class F, class Allocator>
     explicit packaged_task(allocator_arg_t, const Allocator& a, F&& f);
         Requires: INVOKE (f, t1, t2, ..., tN, R), where t1, t2, ..., tN are values of the corresponding
         types in ArgTypes..., shall be a valid expression. Invoking a copy of f shall behave the same as
         invoking f.
3
         Effects: constructs a new packaged_task object with a shared state and initializes the object's stored
         task with std::forward<F>(f). The constructors that take an Allocator argument use it to allocate
         memory needed to store the internal data structures.
4
         Throws: any exceptions thrown by the copy or move constructor of f, or std::bad_alloc if memory
         for the internal data structures could not be allocated.
   packaged_task(packaged_task&& rhs) noexcept;
5
         Effects: constructs a new packaged_task object and transfers ownership of rhs's shared state to
         *this, leaving rhs with no shared state. Moves the stored task from rhs to *this.
6
         Postcondition: rhs has no shared state.
   packaged_task& operator=(packaged_task&& rhs) noexcept;
7
         Effects:
           — releases any shared state (30.6.4).
           — packaged_task(std::move(rhs)).swap(*this).
   ~packaged_task();
         Effects: Abandons any shared state. (30.6.4).
   void swap(packaged_task& other) noexcept;
9
         Effects: exchanges the shared states and stored tasks of *this and other.
10
         Postcondition: *this has the same shared state and stored task (if any) as other prior to the call to
         swap. other has the same shared state and stored task (if any) as *this prior to the call to swap.
   bool valid() const noexcept;
   § 30.6.9.1
                                                                                                      1158
```

11 Returns: true only if \*this has a shared state.

future<R> get\_future();

- Returns: A future object that shares the same shared state as \*this.
- 13 Throws: a future\_error object if an error occurs.
- 14 Error conditions:
  - future\_already\_retrieved if get\_future has already been called on a packaged\_task object with the same shared state as \*this.
  - no\_state if \*this has no shared state.

void operator()(ArgTypes... args);

- Effects: INVOKE (f, t1, t2, ..., tN, R), where f is the stored task of \*this and t1, t2, ..., tN are the values in args.... If the task returns normally, the return value is stored as the asynchronous result in the shared state of \*this, otherwise the exception thrown by the task is stored. The shared state of \*this is made ready, and any threads blocked in a function waiting for the shared state of \*this to become ready are unblocked.
- Throws: a future\_error exception object if there is no shared state or the stored task has already been invoked.
- 17 Error conditions:
  - promise\_already\_satisfied if the stored task has already been invoked.
  - no\_state if \*this has no shared state.
- Synchronization: a successful call to operator() synchronizes with (1.10) a call to any member function of a future or shared\_future object that shares the shared state of \*this. The completion of the invocation of the stored task and the storage of the result (whether normal or exceptional) into the shared state synchronizes with (1.10) the successful return from any member function that detects that the state is set to ready. [Note: operator() synchronizes and serializes with other functions through the shared state. —end note]

void make\_ready\_at\_thread\_exit(ArgTypes... args);

- Effects: INVOKE (f, t1, t2, ..., tN, R), where f is the stored task and t1, t2, ..., tN are the values in args.... If the task returns normally, the return value is stored as the asynchronous result in the shared state of \*this, otherwise the exception thrown by the task is stored. In either case, this shall be done without making that state ready (30.6.4) immediately. Schedules the shared state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.
- 20 Throws: future\_error if an error condition occurs.
- 21 Error conditions:
  - promise already satisfied if the stored task has already been invoked.
  - no\_state if \*this has no shared state.

void reset();

§ 30.6.9.1

```
22
         Effects: as if *this = packaged_task(std::move(f)), where f is the task stored in *this. [Note:
         This constructs a new shared state for *this. The old state is abandoned (30.6.4). — end note]
23
         Throws:
          — bad_alloc if memory for the new shared state could not be allocated.
          — any exception thrown by the move constructor of the task stored in the shared state.
          — future_error with an error condition of no_state if *this has no shared state.
   30.6.9.2 packaged_task globals
                                                                             [futures.task.nonmembers]
   template <class R, class... ArgTypes>
      void swap(packaged_task<R(ArgTypes...)>& x, packaged_task<R(ArgTypes...)>& y) noexcept;
         Effects: x.swap(y)
   template <class R, class Alloc>
      struct uses_allocator<packaged_task<R>, Alloc>
        : true_type { };
2
         Requires: Alloc shall be an Allocator (17.6.3.5).
```

§ 30.6.9.2

# Annex A (informative) Grammar summary

[gram]

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 [gram.key]

New context-dependent keywords are introduced into a program by typedef (7.1.3), namespace (7.3.1), class (clause 9), enumeration (7.2), and template (clause 14) declarations.

```
typedef-name:
    identifier
namespace-name:
    original-namespace-name
namespace-alias

original-namespace-name:
    identifier

namespace-alias:
    identifier
class-name:
    identifier
simple-template-id
enum-name:
    identifier
template-name:
    identifier
template-name:
    identifier
```

Note that a typedef-name naming a class is also a class-name (9.1).

## A.2 Lexical conventions

[gram.lex]

```
hex-quad:
```

```
token:
      identifier
      keyword
      literal
      operator
      punctuator
header-name:
      < h-char-sequence >
      " q-char-sequence "
h-char-sequence:
      h-char
      h\text{-}char\text{-}sequence\ h\text{-}char
h-char:
      any member of the source character set except new-line and >
q	ext{-}char	ext{-}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 \to sign
      pp-number .
identifier:
      identifier\hbox{-} non digit
      identifier\ identifier\ nondigit
      identifier digit
identifier\hbox{-} non digit:
      nondigit
      universal\hbox{-}character\hbox{-}name
      other implementation-defined characters
nondigit: one of
      abcdefghijklm
      nopqrstuvwxyz
      ABCDEFGHIJKLM
      NOPQRSTUVWXYZ_
digit: one of
      0 1 2 3 4 5 6 7 8 9
                           one of
preprocessing-op-or-punc:
      {
            }
                       Γ
                                 ]
                                                       ##
                                                                             )
      <:
            :>
                       <%
                                  %>
                                            %:
                                                       %:%:
                                                                                        . . .
                       ?
            delete
      new
                                  ::
                                            %
                                                                  &₹.
                       <
                                  >
                                                                                        %=
                                             +=
                       |=
                                  <<
                                            >>
                                                       >>=
                                                                  <<=
            &=
                                                                                        !=
                       &&
                                  \Pi
                                                                             ->*
                                                                                        ->
      and
            and_eq
                       bitand
                                 bitor
                                             compl
                                                       not
                                                                  not_eq
      or
            or_eq
                       xor
                                  xor_eq
```

```
literal:
       integer-literal
       character\hbox{-}literal
       floating	ext{-}literal
       string	ext{-}literal
       boolean-literal
       pointer-literal
       user-defined-literal
integer\mbox{-}literal:
       decimal\text{-}literal\ integer\text{-}suffix_{opt}
       octal-literal integer-suffix_{opt}
       hexadecimal-literal integer-suffix_{opt}
decimal \hbox{-} literal \hbox{:}
       nonzero\text{-}digit
       decimal-literal digit
octal-literal:
       0
       octal-literal octal-digit
hexadecimal	ext{-}literal	ext{:}
       0x hexadecimal-digit
       \mathtt{OX}\ hexadecimal	ext{-}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
       abcdef
       ABCDEF
integer-suffix:
       unsigned-suffix\ long-suffix_{opt}
       unsigned\text{-}suffix\ long\text{-}long\text{-}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
character\hbox{-}literal:
       ' c-char-sequence'
       u' c-char-sequence '
       U, c-char-sequence,
       L' c-char-sequence '
c-char-sequence:
       c-char
       c\text{-}char\text{-}sequence\ c\text{-}char
c-char:
       any member of the source character set except
              the single-quote ', backslash \setminus, or new-line character
       escape\hbox{-}sequence
       universal\hbox{-}character\hbox{-}name
```

```
escape\mbox{-}sequence:
       simple\text{-}escape\text{-}sequence
       octal-escape-sequence
       hexa decimal \hbox{-} escape \hbox{-} sequence
simple-escape-sequence: one of
              \"
       \'
                    \?
                           11
                                 \r
              \b \f
                          \n
octal-escape-sequence:
       \ octal-digit
        \ \ \  octal-digit octal-digit
        \ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:
        \x hexadecimal-digit
       hexadecimal-escape-sequence hexadecimal-digit
floating-literal:
       fractional-constant exponent-part_{opt} floating-suffix_{opt}
       digit-sequence exponent-part floating-suffix_{opt}
fractional-constant:
       digit\text{-}sequence_{opt}. digit\text{-}sequence
        digit-sequence .
exponent\mbox{-}part:
       e sign_{opt} digit\text{-}sequence
       \texttt{E}\ sign_{opt}\ digit\text{-}sequence
sign: one of
digit\text{-}sequence:
       digit
       digit-sequence digit
floating-suffix: one of
       f 1 F L
string-literal:
        encoding-prefix_{opt}" s-char-sequence_{opt}"
        encoding-prefix_{opt}R raw-string
encoding-prefix:
       u8
       u
       U
       L
s-char-sequence:
       s-char
       s\text{-}char\text{-}sequence\ s\text{-}char
s-char:
       any member of the source character set except
               the double-quote ", backslash \setminus, or new-line character
        escape\hbox{-}sequence
       universal\hbox{-}character\hbox{-}name
raw-string:
        " d-char-sequence<sub>opt</sub> ( r-char-sequence<sub>opt</sub> ) d-char-sequence<sub>opt</sub> "
r-char-sequence:
       r-char
       r	ext{-}char	ext{-}sequence\ r	ext{-}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\hbox{-}literal:
           false
           true
    pointer-literal:
           nullptr
    user-defined-literal:
           user\hbox{-} defined\hbox{-} integer\hbox{-} literal
           user-defined-floating-literal
           user-defined-string-literal
           user-defined-character-literal
    user\hbox{-} defined\hbox{-} integer\hbox{-} literal:
           decimal-literal ud-suffix
           octal-literal ud-suffix
           hexadecimal-literal ud-suffix
    user-defined-floating-literal:
           fractional\text{-}constant\ exponent\text{-}part_{opt}\ ud\text{-}suffix
           digit\text{-}sequence\ exponent\text{-}part\ ud\text{-}suffix
    user-defined-string-literal:
           string\mbox{-}literal\ ud\mbox{-}suffix
    user-defined\hbox{-}character-literal:
           character\hbox{-}literal\ ud\hbox{-}suffix
    ud-suffix:
           identifier
                                                                                                              [gram.basic]
 Basic concepts
translation\hbox{-}unit:
       declaration\hbox{-}seq_{opt}
                                                                                                               [gram.expr]
 Expressions
primary-expression:
       literal
       this
       ( expression )
       id-expression
       lambda\hbox{-}expression
    id-expression:
           unqualified-id
           qualified\hbox{-} id
```

§ A.4 1165

 $\mathbf{A.3}$ 

```
unqualified-id:
        identifier
        operator\mbox{-}function\mbox{-}id
        conversion\hbox{-} function\hbox{-} id
        literal	ext{-}operator	ext{-}id
        \sim class-name
        \sim decltype-specifier
        template\text{-}id
qualified-id:
        nested-name-specifier template_{opt} unqualified-id
        :: identifier
        :: \ operator\text{-}function\text{-}id
        :: \ literal\hbox{-} operator\hbox{-} id
        :: template-id
nested-name-specifier:
        ::_{opt} \ type-name ::
        ::_{opt} namespace-name::
        decltype\text{-}specifier::
        nested\text{-}name\text{-}specifier\ identifier::
        nested\text{-}name\text{-}specifier \ \mathtt{template}_{opt} \ simple\text{-}template\text{-}id \ :: \\
lambda\mbox{-}expression:
        lambda-introducer lambda-declarator_{opt} compound-statement
lambda\hbox{-}introducer:
        [ lambda-capture_{opt}]
lambda-capture:
        capture-default
        capture\hbox{-}list
        capture\mbox{-}default , capture\mbox{-}list
capture\mbox{-}default:
        &
capture\mbox{-}list:
        capture \dots_{opt}
        capture-list , capture \dots_{opt}
capture:
        identifier
        & identifier
        this
lambda\hbox{-}declarator:
        ( parameter\text{-}declaration\text{-}clause ) \mathtt{mutable}_{opt}
                exception-specification_{opt}\ attribute-specifier-seq_{opt}\ trailing-return-type_{opt}
```

§ A.4 1166

```
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<sub>opt</sub> id-expression
       postfix-expression \rightarrow template<sub>opt</sub> id-expression
       post fix-expression . pseudo-destructor-name
       postfix-expression \rightarrow pseudo-destructor-name
       postfix-expression ++
       postfix-expression --
       dynamic_cast < type-id > (expression)
       static_cast < type-id > ( expression )
      reinterpret_cast < type-id > ( expression )
       const_cast < type-id > ( expression )
       typeid ( expression )
      typeid (type-id)
expression-list:
       initializer-list
pseudo-destructor-name:
       nested-name-specifier_{opt} type-name :: \sim type-name
       nested-name-specifier template simple-template-id :: ~ type-name
       nested-name-specifier_{opt} \sim type-name
       ~ decltype-specifier
unary-expression:
      postfix\mbox{-}expression
      ++ cast-expression
       -- cast-expression
       unary\text{-}operator\ cast\text{-}expression
      {\tt sizeof}\ unary-expression
       sizeof (type-id)
       sizeof ... ( identifier )
       alignof ( type-id )
       no except\hbox{-} expression
       new\text{-}expression
       delete\hbox{-}expression
unary-operator: one of
      * & + - ! ~
new\mbox{-}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	ext{-}specifier	ext{-}seq\ new	ext{-}declarator_{opt}
new	ext{-}declarator:
      ptr-operator new-declarator_{opt}
       noptr-new-declarator
```

§ A.4 1167

```
noptr-new-declarator:
       [ expression ] attribute-specifier-seq_{opt}
       noptr-new-declarator [ constant-expression ] attribute-specifier-seq_{opt}
new\mbox{-}initializer:
       ( expression-list_{opt})
       braced\hbox{-}init\hbox{-}list
delete\mbox{-}expression:
       ::_{opt}delete cast-expression
       :: {}_{opt} \mathtt{delete} [ ] cast\text{-}expression
no except-expression:
       {\tt noexcept} ( {\it expression} )
cast-expression:
       unary-expression
       ( type-id ) cast-expression
pm-expression:
       cast\text{-}expression
       pm-expression .* cast-expression
       pm-expression \rightarrow * cast-expression
multiplicative\mbox{-}expression:
       pm\text{-}expression
       multiplicative\text{-}expression * pm\text{-}expression
       multiplicative-expression / pm-expression
       multiplicative\text{-}expression \ \% \ pm\text{-}expression
additive\-expression:
       multiplicative-expression
       additive\text{-}expression + multiplicative\text{-}expression
       additive\text{-}expression - multiplicative\text{-}expression
shift-expression:
       additive\mbox{-}expression
       shift-expression << additive-expression
       shift-expression >> additive-expression
relational-expression:
       shift-expression
       relational-expression < shift-expression
       relational-expression > shift-expression
       relational-expression \leftarrow shift-expression
       relational-expression >= shift-expression
equality-expression:
       relational-expression
       equality-expression == relational-expression
       equality-expression != relational-expression
and\mbox{-}expression:
       equality\hbox{-}expression
       and-expression & equality-expression
exclusive \hbox{-} or \hbox{-} expression:
       and\mbox{-}expression
       exclusive-or-expression ^ and-expression
inclusive-or-expression:
       exclusive-or-expression
       inclusive-or-expression \mid exclusive-or-expression
logical-and-expression:
       inclusive-or-expression
       logical-and-expression && inclusive-or-expression
```

§ A.4 1168

```
logical-or-expression:
           logical-and-expression
           logical-or-expression || logical-and-expression
    conditional\hbox{-} expression:
           logical \hbox{-} or \hbox{-} expression
           logical-or-expression ? expression : assignment-expression
    assignment\hbox{-} expression:
           conditional-expression
           logical \hbox{-} or \hbox{-} expression \ assignment \hbox{-} operator \ initializer \hbox{-} clause
           throw\mbox{-}expression
    assignment-operator: one of
           = *= /= %= += -= >>= <<= &= ^= |=
    expression:
           assignment\mbox{-}expression
           expression, assignment-expression
    constant-expression:
           conditional\hbox{-} expression
 Statements
                                                                                                           [gram.stmt]
statement:
       labeled\hbox{-} statement
       attribute-specifier-seq_{opt} expression-statement
       attribute-specifier-seq_{opt} compound-statement
       attribute-specifier-seq_{opt} selection-statement
       attribute-specifier-seq_{opt} iteration-statement
       attribute-specifier-seq_{opt} jump-statement
       declaration\hbox{-} statement
       attribute-specifier-seq_{opt} try-block
    labeled\mbox{-}statement:
           attribute-specifier-seq_{opt} identifier: statement
           attribute-specifier-seq_{opt}case constant-expression : statement
           attribute-specifier-seq_{opt}default : statement
    expression\mbox{-}statement:
           expression_{opt};
    compound-statement:
           { statement\text{-}seq_{opt}}
    statement-seq:
           statement
           statement-seq statement
    selection\text{-}statement:
           if (condition) statement
           if (condition) statement else statement
           switch ( condition ) statement
    condition:
           expression
           attribute-specifier-seq_{opt} decl-specifier-seq declarator = initializer-clause
           attribute-specifier-seq<sub>opt</sub> decl-specifier-seq declarator braced-init-list
    iteration\text{-}statement:
           while ( condition ) statement
           do statement while ( expression ) ;
           for (for-init-statement condition<sub>opt</sub>; expression<sub>opt</sub>) statement
           for (for-range-declaration: for-range-initializer) statement
```

A.5

```
for-init-statement:
                     expression\text{-}statement
                     simple-declaration
            for\mbox{-} range\mbox{-} declaration:
                     attribute\text{-}specifier\text{-}seq\ decl-specifier\text{-}seq\ declarator
            for-range-initializer:
                     expression\\
                     braced\hbox{-}init\hbox{-}list
            jump\text{-}statement:
                     break ;
                     continue;
                     return expression_{opt};
                     {\tt return}\ braced{-init-list} ;
                     goto identifier ;
             declaration\text{-}statement:
                     block-declaration
                                                                                                                                         [gram.dcl]
\mathbf{A.6}
          Declarations
        declaration-seq:
                declaration\\
                declaration-seq declaration
             declaration:
                     block\text{-}declaration
                     function\hbox{-} definition
                     template\text{-}declaration
                     explicit\hbox{-}instantiation
                     explicit\text{-}specialization
                     linkage\text{-}specification
                     name space-definition
                     empty\text{-}declaration
                     attribute\text{-}declaration
             block\text{-}declaration:
                     simple-declaration
                     asm-definition
                     name space \hbox{-} alias \hbox{-} definition
                     using\hbox{-}declaration
                     using\hbox{-} directive
                     static\_assert\text{-}declaration
                     a lias\text{-}declaration
                     opaque\text{-}enum\text{-}declaration
             alias-declaration:
                     using identifier\ attribute-specifier-seq_{opt} = type-id;
             simple-declaration:
                     decl\mbox{-specifier-seq}_{opt} init\mbox{-}declarator\mbox{-}list_{opt};
                     attribute\text{-}specifier\text{-}seq\ decl\text{-}specifier\text{-}seq_{opt}\ init\text{-}declarator\text{-}list\ \textbf{;}
             static\_assert\text{-}declaration:
                     static_assert ( constant-expression , string-literal ) ;
             empty\mbox{-}declaration:
             attribute\text{-}declaration:
                     attribute-specifier-seq ;
```

```
decl\mbox{-}specifier:
        storage\text{-}class\text{-}specifier
        type\text{-}specifier
        function\text{-}specifier
        friend
        typedef
        constexpr
decl\text{-}specifier\text{-}seq:
        decl\text{-}specifier\ attribute\text{-}specifier\text{-}seq_{opt}
        decl\text{-}specifier\ decl\text{-}specifier\text{-}seq
storage\text{-}class\text{-}specifier:
        register
        static
        thread_local
        extern
        mutable
function\hbox{-}specifier:
        inline
        virtual
        explicit
typedef-name:
        identifier
type-specifier:
        trailing\hbox{-}type\hbox{-}specifier
        class\mbox{-}specifier
        enum-specifier
trailing\-type\-specifier:
        simple-type-specifier
        elaborated\hbox{-}type\hbox{-}specifier
        typename\text{-}specifier
        cv-qualifier
type\text{-}specifier\text{-}seq:
        type	ext{-}specifier \ attribute	ext{-}specifier	ext{-}seq_{opt}
        type	ext{-}specifier \ type	ext{-}specifier-seq
trailing-type-specifier-seq:\\
        trailing-type-specifier attribute-specifier-seq_{opt}
        trailing-type-specifier\ trailing-type-specifier-seq
```

```
simple-type-specifier:
      nested-name-specifier_{opt} type-name
      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
      simple-template-id
decltype-specifier:
      decltype ( expression )
elaborated-type-specifier:
      class-key\ attribute-specifier-seq_{opt}\ nested-name-specifier_{opt}\ identifier
      class-key\ nested-name-specifier_{opt} {\tt template}_{opt}\ simple-template-id
      enum nested-name-specifier_{opt} identifier
enum-name:
      identifier
enum\text{-}specifier:
      enum-head \{ enumerator-list_{opt} \}
      enum-head { enumerator-list , }
enum-head:
      enum-key attribute-specifier-seq_{opt} identifier_{opt} enum-base_{opt}
      enum-key attribute-specifier-seq<sub>opt</sub> nested-name-specifier identifier
             enum-base_{opt}
opaque-enum-declaration:
      enum-key attribute-specifier-seq_{opt} identifier enum-base_{opt};
enum-key:
      enum
      enum class
      enum struct
enum-base:
      : type-specifier-seq
enumerator-list:
      enumerator\hbox{-} definition
      enumerator\mbox{-}list \ \ , \ \ enumerator\mbox{-}definition
enumerator-definition:
      enumerator
      enumerator = constant\text{-}expression
enumerator:
      identifier
```

```
namespace-name:
      original \hbox{-} name space \hbox{-} name
      name space-alias
original-namespace-name:
      identifier
name space - definition:
      named\hbox{-}name space\hbox{-}definition
      unnamed\text{-}namespace\text{-}definition
named-namespace-definition:
      original-name space-definition
      extension-namespace-definition
original-namespace-definition:
      inline<sub>opt</sub> namespace identifier { namespace-body }
extension-namespace-definition:
      inline_{opt} namespace original-namespace-name { namespace-body }
unnamed-namespace-definition:
      inline_{opt} namespace { namespace-body }
name space-body:
      declaration-seq_{opt}
name space-alias:
      identifier
name space-alias-definition:
      namespace identifier = qualified-namespace-specifier ;
qualified\hbox{-}name space\hbox{-}specifier:
      nested-name-specifier_{opt} namespace-name
using	ext{-}declaration:
      using typename_{opt} nested-name-specifier unqualified-id;
      using :: unqualified-id;
using-directive:
      attribute-specifier-seq_{opt}using namespace nested-name-specifier_{opt} namespace-name;
as m\text{-}definition:
      asm (string-literal);
linkage-specification:
      extern string-literal { declaration-seq<sub>opt</sub>}
      extern string-literal declaration
attribute-specifier-seq:
      attribute-specifier-seq_{opt} attribute-specifier
attribute-specifier:
      [ [ attribute-list ] ]
      alignment\mbox{-}specifier
alignment-specifier:
      alignas ( type\text{-}id \dots _{opt} )
      alignas ( alignment-expression ... opt)
attribute\hbox{-} list:
      attribute_{opt}
      attribute-list, attribute_{opt}
      attribute \dots
      attribute-list, attribute ...
attribute:
      attribute-token attribute-argument-clause<sub>opt</sub>
```

```
attribute-token:
                  identifier
                  attribute\text{-}scoped\text{-}token
           attribute-scoped-token:
                   attribute{-namespace}:: identifier
           attribute-namespace:
                  identifier
           attribute-argument-clause:
                   ( balanced-token-seq )
           balanced\hbox{-}token\hbox{-}seq:
                   balanced-tokenopt
                   balanced\text{-}token\text{-}seq\ balanced\text{-}token
           balance d\hbox{-}token:
                   ( balanced-token-seq )
                   [ balanced-token-seq ]
                   { balanced-token-seq }
                  any token other than a parenthesis, a bracket, or a brace
        Declarators
                                                                                                                       [gram.decl]
A.7
       init-declarator-list:
              init-declarator
              init-declarator-list , init-declarator
           init-declarator:
                   declarator\ initializer_{opt}
           declarator:
                  ptr-declarator
                   noptr-declarator\ parameters-and-qualifiers\ trailing-return-type
           ptr-declarator:
                   noptr	ext{-}declarator
                  ptr-operator ptr-declarator
           noptr\mbox{-}declarator:
                   declarator-id attribute-specifier-seq_{opt}
                   noptr\mbox{-}declarator\ parameters\mbox{-}and\mbox{-}qualifiers
                   noptr-declarator [ constant-expression_{opt}] attribute-specifier-seq_{opt}
                   ( ptr-declarator )
           parameters-and-qualifiers:
                   ( parameter-declaration-clause ) attribute-specifier-seq_{opt} cv-qualifier-seq_{opt}
                          ref-qualifier_{opt} exception-specification_{opt}
           trailing-return-type:
                   -> trailing-type-specifier-seq abstract-declarator<sub>opt</sub>
           ptr-operator:
                  * attribute-specifier-seq_{opt} cv-qualifier-seq_{opt}
                  & attribute-specifier-seq<sub>opt</sub>
                  && attribute-specifier-seq_{opt}
                   nested-name-specifier *\ attribute-specifier-seq_{opt}\ cv-qualifier-seq_{opt}
           cv-qualifier-seq:
                   cv-qualifier cv-qualifier-seq_{opt}
           cv-qualifier:
                  const
                  volatile
           ref-qualifier:
                   &&
```

```
declarator	ext{-}id:
       \dots_{opt} id-expression
       nested-name-specifier_{opt} class-name
type	ext{-}id	ext{:}
       type	ext{-}specifier	ext{-}seq\ abstract	ext{-}declarator_{opt}
abstract-declarator:
       ptr-abstract-declarator
       noptr-abstract-declarator_{opt} parameters-and-qualifiers trailing-return-type
       abstract-pack-declarator
ptr-abstract-declarator:
       noptr-abstract-declarator
       ptr-operator ptr-abstract-declarator_{opt}
noptr-abstract-declarator:
       noptr-abstract-declarator_{opt}\ parameters-and-qualifiers
       noptr-abstract-declarator_{opt}[ constant-expression_{opt}] attribute-specifier-seq_{opt}
       ( ptr-abstract-declarator )
abstract-pack-declarator:
       noptr-abstract-pack-declarator
       ptr-operator abstract-pack-declarator
noptr-abstract-pack-declarator:
       noptr-abstract-pack-declarator parameters-and-qualifiers
       noptr-abstract-pack-declarator [ constant-expression_{opt} ] attribute-specifier-seq_{opt}
parameter-declaration-clause:
       parameter-declaration-list_{opt}..._{opt}
       parameter-declaration-list, ...
parameter-declaration-list:
       parameter-declaration
       parameter-declaration-list \ \ , \ parameter-declaration
parameter-declaration:
       attribute-specifier-seq_{opt} decl-specifier-seq declarator
       attribute\textit{-specifier-seq}\ decl\textit{-specifier-seq}\ declarator = initializer\textit{-clause}
       attribute-specifier-seq abstract-declarator_{opt}
       attribute-specifier-seq_{opt} decl-specifier-seq_{abstract}-declarator_{opt}=_{initializer}-clause
function-definition:
       attribute-specifier-seq_{opt} decl-specifier-seq_{opt} declarator virt-specifier-seq_{opt} function-body
function-body:
       ctor	ext{-}initializer_{opt} compound	ext{-}statement
       function-try-block
       = default ;
       = delete ;
initializer:
       brace\hbox{-} or\hbox{-} equal\hbox{-} initializer
       ( expression-list )
brace-or-equal-initializer:
       = initializer-clause
       braced	ext{-}init	ext{-}list
initializer\mbox{-}clause:
       assignment\text{-}expression
       braced-init-list
```

```
initializer-list:
                   initializer-clause ... opt
                   initializer-list , initializer-clause . . . _{opt}
            braced	ext{-}init	ext{-}list:
                   { initializer-list , _{opt} }
                   { }
A.8
         Classes
                                                                                                                          [gram.class]
       class-name:
               identifier
               simple\mbox{-}template\mbox{-}id
            class\mbox{-}specifier:
                   class-head { member-specification_{opt}}
            class{-}head:
                   class-key\ attribute-specifier-seq_{opt}\ class-head-name\ class-virt-specifier_{opt}\ base-clause_{opt}
                   class-key \ attribute-specifier-seq_{opt} \ base-clause_{opt}
                   nested-name-specifier_{opt} class-name
            class\text{-}virt\text{-}specifier:
                   final
            class\text{-}key:
                   class
                   struct
                   union
            member\mbox{-}specification:
                   member-declaration\ member-specification_{opt}
                   access-specifier: member-specification_{opt}
            member-declaration:
                   attribute-specifier-seq<sub>opt</sub> decl-specifier-seq<sub>opt</sub> member-declarator-list<sub>opt</sub>;
                   function-definition; opt
                   using	ext{-}declaration
                   static \ assert-declaration
                   template\text{-}declaration
                   a lias\text{-}declaration
            member-declarator-list:
                   member-declarator
                   member-declarator-list , member-declarator
            member-declarator:
                   declarator\ virt\text{-}specifier\text{-}seq_{opt}\ pure\text{-}specifier_{opt}
                   declarator\ brace-or-equal-initializer_{opt}
                   identifier_{opt} attribute-specifier-seq_{opt}: constant-expression
            virt-specifier-seq:
                   virt\text{-}specifier
                   virt-specifier-seq virt-specifier
            virt-specifier:
                   override
                   final
            pure-specifier:
\mathbf{A.9}
        Derived classes
                                                                                                                      [gram.derived]
       base-clause:
               : base-specifier-list
```

```
base-specifier-list:
                   \textit{base-specifier} \ \dots_{\textit{opt}}
                   base-specifier-list , base-specifier \dots_{opt}
            base\mbox{-specifier:}
                   attribute\text{-}specifier\text{-}seq_{opt}\ base\text{-}type\text{-}specifier
                   attribute-specifier-seq_{opt}virtual access-specifier_{opt} base-type-specifier
                   attribute-specifier-seq_{opt} access-specifier virtual_{opt} base-type-specifier
            class-or-decltype:
                   nested-name-specifier_{opt} class-name
                   decltype\text{-}specifier
            base-type-specifier:
                   class-or-decltype
            access\hbox{-} specifier:
                   private
                   protected
                   public
           Special member functions
                                                                                                                      [gram.special]
A.10
       conversion-function-id:
               operator conversion-type-id
            conversion\mbox{-}type\mbox{-}id:
                   type-specifier-seq conversion-declarator_{opt}
            conversion\mbox{-}declarator:
                   ptr-operator conversion-declarator opt
            ctor	ext{-}initializer:
                   : mem\mbox{-}initializer\mbox{-}list
            mem-initializer-list:
                   mem-initializer ... _{opt}
                   mem-initializer, mem-initializer-list . . . opt
            mem-initializer:
                   mem-initializer-id ( expression-list_{opt})
                   mem\text{-}initializer\text{-}id\ braced\text{-}init\text{-}list
            mem\text{-}initializer\text{-}id:
                   class-or-decltype
                   identifier
                                                                                                                           [gram.over]
A.11
           Overloading
       operator\mbox{-}function\mbox{-}id:
               operator operator
            operator:
                         one of
                   new
                           delete
                                        new[]
                                                     delete[]
                                                     /
                                                     >
                                                                  +=
                                        |=
                                                                  >>
                                                                               >>=
                                                                                            <<=
                           &=
                                                     <<
                   <=
                                                     \Pi
                           >=
                   ()
                           []
            literal-operator-id:
                   {\tt operator} \verb|""| identifier
A.12
           Templates
                                                                                                                         [gram.temp]
       template\text{-}declaration:
               {\tt template} \, < \, template\text{-}parameter\text{-}list \, > \, declaration
```

```
template-parameter-list:\\
           template-parameter
           template-parameter-list , template-parameter
    template	ext{-}parameter:
           type	ext{-}parameter
           parameter-declaration
    type\mbox{-}parameter:
           class ... _{opt} identifier_{opt}
           class identifier_{opt}= type\text{-}id
           typename ... _{opt} identifier_{opt}
           typename identifier_{opt}= type-id
           template < template-parameter-list > class ... opt identifier_{opt}
           \verb|template| < template-parameter-list| > \verb|class| identifier_{opt} = id\text{-}expression|
    simple-template-id:
           template-name < template-argument-list_{opt}>
    template \hbox{-} id \hbox{:}
           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	ext{-}argument:
           constant\hbox{-} expression
           type-id
           id\mbox{-}expression
    typename-specifier:
           typename nested-name-specifier identifier
           typename nested-name-specifier template<sub>opt</sub> simple-template-id
    explicit	ext{-}instantiation:
           extern_{opt} template declaration
    explicit \hbox{-} specialization:
           {\tt template} \, \mathrel{<} \, \mathrel{>} \, declaration
                                                                                                           [gram.except]
   Exception handling
try-block:
       try compound-statement handler-seq
    function-try-block:
           try ctor-initializer<sub>opt</sub> compound-statement handler-seq
    handler-seq:
           handler handler-seq<sub>opt</sub>
    handler:
           catch ( exception-declaration ) compound-statement
           attribute\text{-}specifier\text{-}seq_{opt}\ type\text{-}specifier\text{-}seq\ declarator
           attribute-specifier-seq abstract-declarator_{opt}
    throw-expression:
           throw assignment-expression_{opt}
```

§ A.13

A.13

 $exception\mbox{-}specification:$ 

```
dynamic \hbox{-} exception \hbox{-} specification
                   no except-specification
            dynamic\mbox{-}exception\mbox{-}specification:
                   throw ( type-id-list_{opt})
            type-id-list:
                   type-id \dots opt
                   type-id-list , type-id \dots_{opt}
            noexcept-specification:
                   noexcept ( constant-expression )
                   noexcept
                                                                                                                           [gram.cpp]
A.14
          Preprocessing directives
       preprocessing-file:
              group_{opt}
            group:
                   group-part
                   group group-part
            group-part:
                   if-section
                   control\mbox{-}line
                   text-line
                   \# non-directive
            if-section:
                   {\it if-group}\ elif-groups_{opt}\ else-group_{opt}\ endif-line
            if-group:
                   # if
                                        constant\mbox{-}expression new\mbox{-}line group_{opt}
                   # ifdef
                                        identifier new-line group<sub>opt</sub>
                   # ifndef
                                        identifier new-line group<sub>opt</sub>
            elif-groups:
                   {\it elif-group}
                   elif	ext{-}groups elif	ext{-}group
            elif-group:
                   # elif
                                        constant-expression new-line group_{opt}
            else-group:
                                        new-line group_{opt}
                   # else
            endif-line:
                   # endif
                                        new-line
            control\mbox{-}line:
                                        pp-tokens new-line
                   # include
                                        identifier\ replacement\text{-}list\ new\text{-}line
                   # define
                   # define
                                        identifier\ lparen\ identifier\ list_{opt})\ replacement\ list\ new\ line
                   # define
                                        identifier\ lparen\ \dots ) replacement-list\ new-line
                   # define
                                        identifier\ lparen\ identifier\ list,\ \dots ) replacement\mbox{-list}\ new\mbox{-line}
                   # undef
                                        identifier\ new\mbox{-}line
                   # line
                                        pp-tokens new-line
                                        pp\text{-}tokens_{opt} new\text{-}line
                   # error
                   # pragma
                                        pp-tokens<sub>opt</sub> new-line
                   \# new-line
            text-line:
                   pp-tokens<sub>opt</sub> new-line
            non\text{-}directive:
                   pp-tokens new-line
```

§ A.14 1179

 $\odot$  ISO/IEC N3337

```
\begin{tabular}{l} lparen: & a ( character not immediately preceded by white-space $identifier-list: $identifier $identifier $identifier $identifier$ \\ $identifier-list $, identifier$ \\ $replacement-list: $pp-tokens_{opt}$ \\ $pp-tokens: $preprocessing-token $pp-tokens $preprocessing-token$ \\ $new-line: $the new-line character$ \\ \end{tabular}
```

# Annex B (informative) Implementation quantities

# [implimits]

- Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process. Every implementation shall document those limitations where known. This documentation may cite fixed limits where they exist, say how to compute variable limits as a function of available resources, or say that fixed limits do not exist or are unknown.
- The limits may constrain quantities that include those described below or others. The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine compliance.
  - Nesting levels of compound statements, iteration control structures, and selection control structures [256].
  - Nesting levels of conditional inclusion [256].
  - Pointer, array, and function declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration [256].
  - Nesting levels of parenthesized expressions within a full-expression [256].
  - Number of characters in an internal identifier or macro name [1024].
  - Number of characters in an external identifier [1024].
  - External identifiers in one translation unit [65 536].
  - Identifiers with block scope declared in one block [1024].
  - Macro identifiers simultaneously defined in one translation unit [65 536].
  - Parameters in one function definition [256].
  - Arguments in one function call [256].
  - Parameters in one macro definition [256].
  - Arguments in one macro invocation [256].
  - Characters in one logical source line [65 536].
  - Characters in a string literal (after concatenation) [65 536].
  - Size of an object [262 144].
  - Nesting levels for #include files [256].
  - Case labels for a switch statement (excluding those for any nested switch statements) [16 384].
  - Data members in a single class [16 384].
  - Enumeration constants in a single enumeration [4096].
  - Levels of nested class definitions in a single member-specification [256].

- Functions registered by atexit() [32].
- Functions registered by at\_quick\_exit() [32].
- Direct and indirect base classes [16 384].
- 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 [6 144].
- 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 (20.8.9.1.3) [10].

# Annex C (informative) Compatibility

[diff]

 $C.1 \quad C \mapsto \text{ and ISO } C$  [diff.iso]

This subclause lists the differences between C++ and ISO C, by the chapters of this document.

#### C.1.1 Clause 2: lexical conventions

[diff.lex]

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:

```
int function( int i );
int function( char c );
function( 'x' );
```

It is preferable that this call match the second version of function rather than the first.

Effect on original feature: Change to semantics of well-defined feature. ISO C programs which depend on

```
sizeof('x') == sizeof(int)
```

will not work the same as C++ programs.

Difficulty of converting: Simple.

How widely used: Programs which depend upon sizeof('x') are probably rare.

```
Subclause 2.14.5:
```

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:

**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.

Change: C++ does not have "tentative definitions" as in C E.g., at file scope,

```
int i;
int i;
```

is valid in C, invalid in C++. This makes it impossible to define mutually referential file-local static objects, if initializers are restricted to the syntactic forms of C. For example,

```
struct X { int i; struct X *next; };
static struct X a;
static struct X b = { 0, &a };
static struct X a = { 1, &b };
```

Rationale: This avoids having different initialization rules for fundamental types and user-defined types.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation.

**Rationale:** In C++, the initializer for one of a set of mutually-referential file-local static objects must invoke a function call to achieve the initialization.

How widely used: Seldom.

3.3

Change: A struct is a scope in C++, not in C

Rationale: Class scope is crucial to C++, and a struct is a class.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation.

How widely used: C programs use struct extremely frequently, but the change is only noticeable when struct, enumeration, or enumerator names are referred to outside the struct. The latter is probably rare. 3.5 [also 7.1.6]

Change: A name of file scope that is explicitly declared const, and not explicitly declared extern, has internal linkage, while in C it would have external linkage

Rationale: Because const objects can be used as compile-time values in C++, this feature urges programmers to provide explicit initializer values for each const. This feature allows the user to put constobjects in header files that are included in many compilation units.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation

How widely used: Seldom

3.6

Change: Main cannot be called recursively and cannot have its address taken

Rationale: The main function may require special actions.

Effect on original feature: Deletion of semantically well-defined feature

Difficulty of converting: Trivial: create an intermediary function such as mymain(argc, argv).

How widely used: Seldom

3.9

Change: C allows "compatible types" in several places, C++ does not For example, otherwise-identical

struct types with different tag names are "compatible" in C but are distinctly different types in C++.

Rationale: Stricter type checking is essential for C++.

Effect on original feature: Deletion of semantically well-defined feature.

**Difficulty of converting:** Semantic transformation. The "typesafe linkage" mechanism will find many, but not all, of such problems. Those problems not found by typesafe linkage will continue to function properly, according to the "layout compatibility rules" of this International Standard.

How widely used: Common.

#### C.1.3 Clause 4: standard conversions

[diff.conv]

4.10

Change: Converting void\* to a pointer-to-object type requires casting

```
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:

```
char *c = (char *) b;
```

**How widely used:** This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

4.10

Change: Only pointers to non-const and non-volatile objects may be implicitly converted to void\* Rationale: This improves type safety.

Effect on original feature: Deletion of semantically well-defined feature.

**Difficulty of converting:** Could be automated. A C program containing such an implicit conversion from, e.g., pointer-to-const-object to void\* will receive a diagnostic message. The correction is to add an explicit cast.

How widely used: Seldom.

# C.1.4 Clause 5: expressions

[diff.expr]

5.2.2

Change: Implicit declaration of functions is not allowed

**Rationale:** The type-safe nature of C++.

**Effect on original feature:** Deletion of semantically well-defined feature. Note: the original feature was labeled as "obsolescent" in ISO C.

**Difficulty of converting:** Syntactic transformation. Facilities for producing explicit function declarations are fairly widespread commercially.

How widely used: Common.

```
5.3.3, 5.4
```

**Change:** Types must be declared in declarations, not in expressions In C, a size of expression or cast expression may create a new type. For example,

```
p = (void*)(struct x {int i;} *)0;
```

declares a new type, struct x.

Rationale: This prohibition helps to clarify the location of declarations in the source code.

Effect on original feature: Deletion of a semantically well-defined feature.

**Difficulty of converting:** Syntactic transformation.

How widely used: Seldom.

```
5.16, 5.17, 5.18
```

Change: The result of a conditional expression, an assignment expression, or a comma expression may be an lyalue

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 lyalue-to-ryalue conversions will yield different results. For example,

```
char arr[100];
sizeof(0, arr)
```

yields 100 in C++ and sizeof(char\*) in C.

Difficulty of converting: Programs must add explicit casts to the appropriate rvalue.

How widely used: Rare.

# C.1.5 Clause 6: statements

[diff.stat]

```
6.4.2, 6.6.4
```

Change: It is now invalid to jump past a declaration with explicit or implicit initializer (except across entire block not entered)

Rationale: Constructors used in initializers may allocate resources which need to be de-allocated upon leaving the block. Allowing jump past initializers would require complicated run-time determination of allocation. Furthermore, any use of the uninitialized object could be a disaster. With this simple compile-time rule, C++ assures that if an initialized variable is in scope, then it has assuredly been initialized.

Effect on original feature: Deletion of semantically well-defined feature.

**Difficulty of converting:** Semantic transformation.

How widely used: Seldom.

```
6.6.3
```

**Change:** It is now invalid to return (explicitly or implicitly) from a function which is declared to return a value without actually returning a value

Rationale: The caller and callee may assume fairly elaborate return-value mechanisms for the return of class objects. If some flow paths execute a return without specifying any value, the implementation must embody many more complications. Besides, promising to return a value of a given type, and then not returning such a value, has always been recognized to be a questionable practice, tolerated only because very-old C had no distinction between void functions and int functions.

Effect on original feature: Deletion of semantically well-defined feature.

**Difficulty of converting:** Semantic transformation. Add an appropriate return value to the source code, such as zero.

**How widely used:** Seldom. For several years, many existing C implementations have produced warnings in this case.

#### C.1.6 Clause 7: declarations

[diff.dcl]

#### 7.1.1

Change: In C++, the static or extern specifiers can only be applied to names of objects or functions Using these specifiers with type declarations is illegal in C++. In C, these specifiers are ignored when used on type declarations.

Example:

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:

Rationale: For ease of use, C++ doesn't require that a type name be prefixed with the keywords class, struct or union when used in object declarations or type casts.

Example:

```
class name { /*...*/ }; name i; //i has\ type class name
```

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. One of the 2 types has to be renamed.

How widely used: Seldom.

```
7.1.6 [see also 3.5]
```

Change: const objects must be initialized in C++ but can be left uninitialized in C

Rationale: A const object cannot be assigned to so it must be initialized to hold a useful value.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation.

How widely used: Seldom.

```
7.1.6
```

Change: Banning implicit int

In C++ a decl-specifier-seq must contain a type-specifier, unless it is followed by a declarator for a constructor, a destructor, or a conversion function. In the following example, the left-hand column presents valid C; the right-hand column presents equivalent C++:

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.

```
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:

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:

**Rationale:** In C++, an enumeration is a distinct type.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation.

**How widely used:** Seldom. The only time this affects existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

# C.1.7 Clause 8: declarators

[diff.decl]

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:

**Rationale:** This is to avoid erroneous function calls (i.e., function calls with the wrong number or type of arguments).

Effect on original feature: Change to semantics of well-defined feature. This feature was marked as

"obsolescent" in C.

**Difficulty of converting:** Syntactic transformation. The function declarations using C incomplete declaration style must be completed to become full prototype declarations. A program may need to be updated further if different calls to the same (non-prototype) function have different numbers of arguments or if the type of corresponding arguments differed.

How widely used: Common.

```
8.3.5 [see 5.3.3]
```

Change: In C++, types may not be defined in return or parameter types. In C, these type definitions are allowed

Example:

```
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 potential for major catastrophe.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. The arrays must be declared one element bigger to contain the string terminating '\0'.

**How widely used:** Seldom. This style of array initialization is seen as poor coding style.

#### C.1.8 Clause 9: classes

[diff.class]

```
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 fundamental types. The advantages of the new name space definition were judged to outweigh by far the incompatibility with C described above.

Effect on original feature: Change to semantics of well-defined feature.

**Difficulty of converting:** Semantic transformation. If the hidden name that needs to be accessed is at global scope, the :: C++ operator can be used. If the hidden name is at block scope, either the type or the struct tag has to be renamed.

How widely used: Seldom.

9.7

**Change:** In C++, the name of a nested class is local to its enclosing class. In C the name of the nested class belongs to the same scope as the name of the outermost enclosing class.

Example:

```
struct X {
   struct Y { /* ... */ } y;
};
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:

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:

```
typedef int I;
struct S {
```

Rationale: When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.

Effect on original feature: Deletion of semantically well-defined feature.

**Difficulty of converting:** Semantic transformation. Either the type or the struct member has to be renamed.

How widely used: Seldom.

# C.1.9 Clause 12: special member functions

[diff.special]

12.8

Change: Copying volatile objects

The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:

Rationale: Several alternatives were debated at length. Changing the parameter to volatile const X& would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. If volatile semantics are required for the copy, a user-declared constructor or assignment must be provided. [Note: This user-declared constructor may be explicitly defaulted.  $-end\ note$ ] If non-volatile semantics are required, an explicit const\_cast can be used.

How widely used: Seldom.

#### C.1.10 Clause 16: preprocessing directives

[diff.cpp]

16.8

Change: Whether \_ \_ STDC \_ \_ is defined and if so, what its value is, are implementation-defined

Rationale: C++ is not identical to ISO C. Mandating that \_\_STDC\_\_ be defined would require that translators make an incorrect claim. Each implementation must choose the behavior that will be most useful to its marketplace.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation.

**How widely used:** Programs and headers that reference \_\_STDC \_ are quite common.

```
C.2 C++ and ISO C++ 2003
```

[diff.cpp03]

This subclause lists the differences between C++ and ISO C++ 2003 (ISO/IEC 14882:2003, Programming Languages —  $C^{++}$ ), by the chapters of this document.

# C.2.1 Clause 2: lexical conventions

[diff.cpp03.lex]

2.5

Change: New kinds of string literals Rationale: Required for new features.

Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this International Standard. Specifically, macros named R, u8, u8R, u, uR, U, UR, or LR will not be expanded when adjacent to a string literal but will be interpreted as part of the string literal. For example,

```
#define u8 "abc"
const char *s = u8"def";  // Previously "abcdef", now "def"
2.5
```

Change: User-defined literal string support

Rationale: Required for new features.

Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this International Standard, as the following example illustrates.

Previously, #1 would have consisted of two separate preprocessing tokens and the macro  $\_x$  would have been expanded. In this International Standard, #1 consists of a single preprocessing tokens, so the macro is not expanded.

2.12

Change: New keywords

Rationale: Required for new features.

Effect on original feature: Added to Table 4, the following identifiers are new keywords: alignas, alignof, char16\_t, char32\_t, constexpr, decltype, noexcept, nullptr, static\_assert, and thread\_local. Valid C++ 2003 code using these identifiers is invalid in this International Standard.

2.14.2

Change: Type of integer literals Rationale: C99 compatibility.

Effect on original feature: Certain integer literals larger than can be represented by long could change from an unsigned integer type to signed long long.

# C.2.2 Clause 5: expressions

[diff.cpp03.expr]

5.6

Change: Specify rounding for results of integer / and %

Rationale: Increase portability, C99 compatibility.

Effect on original feature: Valid C++ 2003 code that uses integer division rounds the result toward 0 or toward negative infinity, whereas this International Standard always rounds the result toward 0.

# C.2.3 Clause 7: declarations

[diff.cpp03.dcl.dcl]

7.1

Change: Remove auto as a storage class specifier

Rationale: New feature.

Effect on original feature: Valid C++ 2003 code that uses the keyword auto as a storage class specifier may be invalid in this International Standard. In this International Standard, auto indicates that the type of a variable is to be deduced from its initializer expression.

#### C.2.4 Clause 8: declarators

[diff.cpp03.dcl.decl]

8.5.4

Change: Narrowing restrictions in aggregate initializers

Rationale: Catches bugs.

Effect on original feature: Valid C++ 2003 code may fail to compile in this International Standard. For example, the following code is valid in C++ 2003 but invalid in this International Standard because double to int is a narrowing conversion:

```
int x[] = { 2.0 };
```

# C.2.5 Clause 12: special member functions

[diff.cpp03.special]

12.1, 12.4, 12.8

Change: Implicitly-declared special member functions are defined as deleted when the implicit definition would have been ill-formed.

Rationale: Improves template argument deduction failure.

Effect on original feature: A valid C++ 2003 program that uses one of these special member functions in a context where the definition is not required (e.g., in an expression that is not potentially evaluated) becomes ill-formed.

12.4 (destructors)

Change: User-declared destructors have an implicit exception specification.

Rationale: Clarification of destructor requirements.

Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard. In particular, destructors that throw exceptions will call std::terminate() (without calling std::unexpected()) if their exception specification is noexcept or noexcept(true). For a throwing virtual destructor of a derived class, std::terminate() can be avoided only if the base class virtual destructor has an exception specification that is not noexcept and not noexcept(true).

# C.2.6 Clause 14: templates

[diff.cpp03.temp]

14.1

Change: Remove export

Rationale: No implementation consensus.

Effect on original feature: A valid C++ 2003 declaration containing export is ill-formed in this International Standard.

14.3

Change: Remove whitespace requirement for nested closing template right angle brackets

Rationale: Considered a persistent but minor annoyance. Template aliases representing nonclass types would exacerbate whitespace issues.

Effect on original feature: Change to semantics of well-defined expression. A valid C++ 2003 expression containing a right angle bracket (">") followed immediately by another right angle bracket may now be treated as closing two templates. For example, the following code is valid in C++ 2003 because ">" is a right-shift operator, but invalid in this International Standard because ">" closes two templates.

```
template <class T> struct X { }; template <int N> struct Y { }; X< Y< 1 >> 2 > > x; 14.6.4.2
```

Change: Allow dependent calls of functions with internal linkage

Rationale: Overly constrained, simplify overload resolution rules.

Effect on original feature: A valid C++ 2003 program could get a different result than this International Standard.

# C.2.7 Clause 17: library introduction

[diff.cpp03.library]

17 - 30

Change: New reserved identifiers

Rationale: Required by new features.

Effect on original feature: Valid C++ 2003 code that uses any identifiers added to the C++ standard library by this International Standard may fail to compile or produce different results in This International Standard. A comprehensive list of identifiers used by the C++ standard library can be found in the Index of Library Names in this International Standard.

17.6.1.2

Change: New headers

Rationale: New functionality.

Effect on original feature: The following C++ headers are new: <array>, <atomic>, <chrono>, <codecvt>, <condition\_variable>, <forward\_list>, <future>, <initializer\_list>, <mutex>, <random>, <ratio>, <regex>, <scoped\_allocator>, <system\_error>, <thread>, <tuple>, <typeindex>, <type\_traits>, <unordered\_map>, and <unordered\_set>. In addition the following C compatibility headers are new: <ccomplex>, <cfenv>, <cinttypes>, <cstdalign>, <cstdbool>, <cstdint>, <ctgmath>, and <cuchar>. Valid C++ 2003 code that #includes headers with these names may be invalid in this International Standard.

17.6.3.2

Effect on original feature: Function swap moved to a different header

Rationale: Remove dependency on <algorithm> for swap.

Effect on original feature: Valid C++ 2003 code that has been compiled expecting swap to be in <algorithm> may have to instead include <utility>.

17.6.4.2.2

Change: New reserved namespace

Rationale: New functionality.

Effect on original feature: The global namespace posix is now reserved for standardization. Valid C++ 2003 code that uses a top-level namespace posix may be invalid in this International Standard.

17.6.5.3

Change: Additional restrictions on macro names

Rationale: Avoid hard to diagnose or non-portable constructs.

Effect on original feature: Names of attribute identifiers may not be used as macro names. Valid C++ 2003 code that defines override, final, carries\_dependency, or noreturn as macros is invalid in this International Standard.

# C.2.8 Clause 18: language support library

[diff.cpp03.language.support]

18.6.1.1

Change: Linking new and delete operators

Rationale: The two throwing single-object signatures of operator new and operator delete are now specified to form the base functionality for the other operators. This clarifies that replacing just these two signatures changes others, even if they are not explicitly changed.

Effect on original feature: Valid C++ 2003 code that replaces global new or delete operators may execute differently in this International Standard. For example, the following program should write "custom deallocation" twice, once for the single-object delete and once for the array delete.

```
#include <cstdio>
#include <cstdlib>
#include <new>
void* operator new(std::size_t size) throw(std::bad_alloc) {
  return std::malloc(size);
void operator delete(void* ptr) throw() {
 std::puts("custom deallocation");
  std::free(ptr);
}
int main() {
  int* i = new int;
 delete i;
                                 // single-object delete
  int* a = new int[3];
  delete [] a;
                                 // array delete
 return 0;
```

}

#### 18.6.1.1

Change: operator new may throw exceptions other than std::bad alloc

Rationale: Consistent application of noexcept.

Effect on original feature: Valid C++ 2003 code that assumes that global operator new only throws std::bad\_alloc may execute differently in this International Standard.

# C.2.9 Clause 19: diagnostics library

[diff.cpp03.diagnostics]

19.4

Change: Thread-local error numbers

Rationale: Support for new thread facilities.

Effect on original feature: Valid but implementation-specific C++ 2003 code that relies on errno being the same across threads may change behavior in this International Standard.

# C.2.10 Clause 20: general utilities library

[diff.cpp03.utilities]

20.6.4

Change: Minimal support for garbage-collected regions

Rationale: Required by new feature.

Effect on original feature: Valid C++ 2003 code, compiled without traceable pointer support, that interacts with newer C++ code using regions declared reachable may have different runtime behavior.

```
20.8.3, 20.8.4, 20.8.5, 20.8.6, 20.8.7, 20.8.8
```

Change: Standard function object types no longer derived from std::unary\_function or std::binary\_-

function

Rationale: Superseded by new feature.

Effect on original feature: Valid C++ 2003 code that depends on function object types being derived from unary\_function or binary\_function will execute differently in this International Standard.

# C.2.11 Clause 21: strings library

[diff.cpp03.strings]

21.3

Change: basic string requirements no longer allow reference-counted strings

Rationale: Invalidation is subtly different with reference-counted strings. This change regularizes behavior for this International Standard.

Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard.

Change: Loosen basic string invalidation rules

Rationale: Allow small-string optimization.

Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard. Some const member functions, such as data and c\_str, no longer invalidate iterators.

#### C.2.12 Clause 23: containers library

[diff.cpp03.containers]

23.2

Change: Complexity of size() member functions now constant

Rationale: Lack of specification of complexity of size() resulted in divergent implementations with inconsistent performance characteristics.

Effect on original feature: Some container implementations that conform to C++ 2003 may not conform to the specified size() requirements in this International Standard. Adjusting containers such as std::list to the stricter requirements may require incompatible changes.

23.2

Change: Requirements change: relaxation

Rationale: Clarification.

Effect on original feature: Valid C++ 2003 code that attempts to meet the specified container requirements may now be over-specified. Code that attempted to be portable across containers may need to be adjusted as follows:

```
— not all containers provide size(); use empty() instead of size() == 0;
```

- not all containers are empty after construction (array);
- not all containers have constant complexity for swap() (array).

23.2

Change: Requirements change: default constructible

Rationale: Clarification of container requirements.

Effect on original feature: Valid C++ 2003 code that attempts to explicitly instantiate a container using a user-defined type with no default constructor may fail to compile.

```
23.2.3, 23.2.4
```

Change: Signature changes: from void return types

Rationale: Old signature threw away useful information that may be expensive to recalculate.

Effect on original feature: The following member functions have changed:

```
— erase(iter) for set, multiset, map, multimap
```

- erase(begin, end) for set, multiset, map, multimap
- insert(pos, num, val) for vector, deque, list, forward\_list
- insert(pos, beg, end) for vector, deque, list, forward\_list

Valid C++ 2003 code that relies on these functions returning void (e.g., code that creates a pointer to member function that points to one of these functions) will fail to compile with this International Standard. 23.2.3, 23.2.4

Change: Signature changes: from iterator to const\_iterator parameters

**Rationale:** Overspecification. *Effects:* The signatures of the following member functions changed from taking an iterator to taking a const\_iterator:

```
— insert(iter, val) for vector, deque, list, set, multiset, map, multimap
```

- insert(pos, beg, end) for vector, deque, list, forward\_list
- erase(iter) for set, multiset, map, multimap
- erase(begin, end) for set, multiset, map, multimap
- all forms of list::splice
- all forms of list::merge

Valid C++ 2003 code that uses these functions may fail to compile with this International Standard. 23.2.3, 23.2.4

Change: Signature changes: resize

Rationale: Performance, compatibility with move semantics.

Effect on original feature: For vector, deque, and list the fill value passed to resize is now passed by reference instead of by value, and an additional overload of resize has been added. Valid C++ 2003 code that uses this function may fail to compile with this International Standard.

# C.2.13 Clause 25: algorithms library

[diff.cpp03.algorithms]

25.1

**Change:** Result state of inputs after application of some algorithms

Rationale: Required by new feature.

Effect on original feature: A valid C++ 2003 program may detect that an object with a valid but unspecified state has a different valid but unspecified state with this International Standard. For example, std::remove and std::remove\_if may leave the tail of the input sequence with a different set of values than previously.

# C.2.14 Clause 26: numerics library

[diff.cpp03.numerics]

26.4

Change: Specified representation of complex numbers

Rationale: Compatibility with C99.

Effect on original feature: Valid C++ 2003 code that uses implementation-specific knowledge about the binary representation of the required template specializations of std::complex may not be compatible with this International Standard.

# C.2.15 Clause 27: Input/output library

[diff.cpp03.input.output]

```
27.7.2.1.3, 27.7.3.4, 27.5.5.4
```

Change: Specify use of explicit in existing boolean conversion operators

Rationale: Clarify intentions, avoid workarounds.

Effect on original feature: Valid C++ 2003 code that relies on implicit boolean conversions will fail to compile with this International Standard. Such conversions occur in the following conditions:

- passing a value to a function that takes an argument of type bool;
- using operator== to compare to false or true;
- returning a value from a function with a return type of bool;
- initializing members of type bool via aggregate initialization;
- initializing a const bool& which would bind to a temporary.

#### 27.5.3.1.1

Change: Change base class of std::ios\_base::failure

Rationale: More detailed error messages.

Effect on original feature: std::ios\_base::failure is no longer derived directly from std::exception, but is now derived from std::system\_error, which in turn is derived from std::runtime\_error. Valid C++ 2003 code that assumes that std::ios\_base::failure is derived directly from std::exception may execute differently in this International Standard.

27.5.3

Change: Flag types in std::ios\_base are now bitmasks with values defined as constexpr static members Rationale: Required for new features.

Effect on original feature: Valid C++ 2003 code that relies on std::ios\_base flag types being represented as std::bitset or as an integer type may fail to compile with this International Standard. For example:

# C.3 C standard library

[diff.library]

<sup>1</sup> 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).

- <sup>2</sup> The C++ standard library provides 57 standard macros from the C library, as shown in Table 149.
- 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	1/10	<ul> <li>Standard</li> </ul>	macros
rane	149	— этапцаго	macros

assert	HUGE_VAL	NULL <cstring></cstring>	SIGINT	va_end
BUFSIZ	LC_ALL	NULL <ctime></ctime>	SIGSEGV	va_start
CLOCKS_PER_SEC	LC_COLLATE	NULL <cwchar></cwchar>	SIGTERM	WCHAR_MAX
EDOM	LC_CTYPE	offsetof	SIG_DFL	WCHAR_MIN
EILSEQ	LC_MONETARY	RAND_MAX	SIG_ERR	WEOF <cwchar></cwchar>
EOF	LC_NUMERIC	SEEK_CUR	$\mathtt{SIG\_IGN}$	WEOF <cwctype></cwctype>
ERANGE	LC_TIME	SEEK_END	stderr	_IOFBF
errno	$L_{ t mpnam}$	SEEK_SET	stdin	_IOLBF
EXIT_FAILURE	MB_CUR_MAX	setjmp	stdout	_IONBF
EXIT_SUCCESS	NULL <clocale></clocale>	SIGABRT	TMP_MAX	
FILENAME_MAX	NULL <cstddef></cstddef>	SIGFPE	va_arg	
FOPEN_MAX	NULL <cstdlib></cstdlib>	SIGILL	va_copy	

<sup>4</sup> The C++ standard library provides 57 standard values from the C library, as shown in Table 150.

Table 150 — Standard values

CHAR_BIT	FLT_DIG	INT_MIN	MB_LEN_MAX
CHAR_MAX	FLT_EPSILON	LDBL_DIG	SCHAR_MAX
CHAR_MIN	FLT_MANT_DIG	LDBL_EPSILON	SCHAR_MIN
DBL_DIG	FLT_MAX	LDBL_MANT_DIG	SHRT_MAX
DBL_EPSILON	FLT_MAX_10_EXP	LDBL_MAX	SHRT_MIN
DBL_MANT_DIG	FLT_MAX_EXP	LDBL_MAX_10_EXP	UCHAR_MAX
DBL_MAX	FLT_MIN	LDBL_MAX_EXP	UINT_MAX
DBL_MAX_10_EXP	FLT_MIN_10_EXP	LDBL_MIN	ULONG_MAX
DBL_MAX_EXP	FLT_MIN_EXP	LDBL_MIN_10_EXP	USRT_MAX
DBL_MIN	FLT_RADIX	LDBL_MIN_EXP	
DBL_MIN_10_EXP	FLT_ROUNDS	LONG_MAX	
DBL_MIN_EXP	INT_MAX	LONG_MIN	

- <sup>5</sup> The C++ standard library provides 20 standard types from the C library, as shown in Table 151.
- <sup>6</sup> The C++ standard library provides 2 standard structs from the C library, as shown in Table 152.
- 7 The C++ standard library provides 209 standard functions from the C library, as shown in Table 153.

# C.3.1 Modifications to headers

[diff.mods.to.headers]

<sup>1</sup> For compatibility with the Standard C library, the C++ standard library provides the 18 C headers (D.5),

§ C.3.1

Table 151 — Standard types

clock_t	ldiv_t	size_t <cstdio></cstdio>	va_list
div_t	mbstate_t	size_t <cstdlib></cstdlib>	wctrans_t
FILE	ptrdiff_t	size_t <cstring></cstring>	wctype_t
fpos_t	sig_atomic_t	size_t <ctime></ctime>	wint_t <cwchar></cwchar>
jmp_buf	size_t <cstddef></cstddef>	time_t	<pre>wint_t <cwctype></cwctype></pre>

Table 152 — Standard structs

lconv tm

but their use is deprecated in C++.

#### C.3.2 Modifications to definitions

[diff.mods.to.definitions]

# C.3.2.1 Types char16\_t and char32\_t

[diff.char16]

<sup>1</sup> The types char16\_t and char32\_t are distinct types rather than typedefs to existing integral types.

# C.3.2.2 Type wchar\_t

diff.wchar

wchar\_t is a keyword in this International Standard (2.12). It does not appear as a type name defined in any of <cstddef>, <cstdlib>, or <cwchar> (21.7).

#### C.3.2.3 Header <iso646.h>

[diff.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.3.2.4 Macro NULL

[diff.null]

The macro NULL, defined in any of <clocale>, <cstddef>, <cstdio>, <cstdlib>, <cstring>, <ctime>, or <cwchar>, is an implementation-defined C++ null pointer constant in this International Standard (18.2).

#### C.3.3 Modifications to declarations

[diff.mods.to.declarations]

- <sup>1</sup> Header <cstring>: The following functions have different declarations:
  - strchr
  - strpbrk
  - strrchr
  - strstr
  - memchr

21.7 describes the changes.

# C.3.4 Modifications to behavior

[diff.mods.to.behavior]

- <sup>1</sup> Header <cstdlib>: The following functions have different behavior:
  - atexit
  - exit
  - abort

18.5 describes the changes.

<sup>2</sup> Header <csetjmp>: The following functions have different behavior:

§ C.3.4

 $\odot$  ISO/IEC N3337

Table 153 — Standard functions

abort	fmod	isupper	mktime	strftime	wcrtomb
abs	fopen	iswalnum	modf	strlen	wcscat
acos	fprintf	iswalpha	perror	strncat	wcschr
asctime	fputc	iswcntrl	pow	$\operatorname{\mathtt{strncmp}}$	wcscmp
asin	fputs	iswctype	printf	strncpy	wcscoll
atan	fputwc	iswdigit	putc	strpbrk	wcscpy
atan2	fputws	iswgraph	${ t putchar}$	strrchr	wcscspn
atexit	fread	iswlower	puts	strspn	wcsftime
atof	free	iswprint	putwc	strstr	wcslen
atoi	freopen	iswpunct	putwchar	strtod	wcsncat
atol	frexp	iswspace	qsort	strtok	wcsncmp
bsearch	fscanf	iswupper	raise	strtol	wcsncpy
btowc	fseek	iswxdigit	rand	strtoul	wcspbrk
calloc	fsetpos	isxdigit	realloc	strxfrm	wcsrchr
ceil	ftell	labs	remove	swprintf	wcsrtombs
clearerr	fwide	ldexp	rename	swscanf	wcsspn
clock	fwprintf	ldiv	rewind	system	wcsstr
cos	fwrite	localeconv	scanf	tan	wcstod
cosh	fwscanf	localtime	setbuf	tanh	wcstok
ctime	getc	log	setlocale	time	wcstol
difftime	getchar	log10	setvbuf	tmpfile	wcstombs
div	getenv	longjmp	signal	tmpnam	wcstoul
exit	gets	malloc	sin	tolower	wcsxfrm
exp	getwc	mblen	sinh	toupper	wctob
fabs	getwchar	mbrlen	sprintf	towctrans	wctomb
fclose	gmtime	mbrtowc	sqrt	towlower	wctrans
feof	isalnum	mbsinit	srand	towupper	wctype
ferror	isalpha	mbsrtowcs	sscanf	${\tt ungetc}$	wmemchr
fflush	iscntrl	mbstowcs	strcat	ungetwc	wmemcmp
fgetc	isdigit	mbtowc	strchr	vfprintf	wmemcpy
fgetpos	isgraph	memchr	strcmp	vfwprintf	wmemmove
fgets	islower	memcmp	strcoll	vprintf	wmemset
fgetwc	isprint	memcpy	strcpy	vsprintf	wprintf
fgetws	ispunct	memmove	strcspn	vswprintf	wscanf
floor	isspace	memset	strerror	vwprintf	

§ C.3.4

— longjmp

18.10 describes the changes.

# C.3.4.1 Macro offsetof(type, member-designator)

[diff.offsetof]

<sup>1</sup> The macro offsetof, defined in <cstddef>, accepts a restricted set of type arguments in this International Standard. 18.2 describes the change.

# C.3.4.2 Memory allocation functions

[diff.malloc]

<sup>1</sup> The functions calloc, malloc, and realloc are restricted in this International Standard. 20.6.13 describes the changes.

§ C.3.4.2

# Annex D (normative) Compatibility features

[depr]

<sup>1</sup> This Clause describes features of the C++ Standard that are specified for compatibility with existing implementations.

<sup>2</sup> 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

[depr.incr.bool]

The use of an operand of type bool with the ++ operator is deprecated (see 5.3.2 and 5.2.6).

# D.2 register keyword

[depr.register]

The use of the register keyword as a *storage-class-specifier* (7.1.1) is deprecated.

# D.3 Implicit declaration of copy functions

[depr.impldec]

The implicit definition of a copy constructor as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. The implicit definition of a copy assignment operator as defaulted is deprecated if the class has a user-declared copy constructor or a user-declared destructor (12.4, 12.8). In a future revision of this International Standard, these implicit definitions could become deleted (8.4).

# D.4 Dynamic exception specifications

[depr.except.spec]

The use of *dynamic-exception-specifications* is deprecated.

# D.5 C standard library headers

[depr.c.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 154.

Table 154 — C headers

<assert.h></assert.h>	<pre><inttypes.h></inttypes.h></pre>	<signal.h></signal.h>	<stdio.h></stdio.h>	<wchar.h></wchar.h>
<pre><complex.h></complex.h></pre>	<iso646.h></iso646.h>	<stdalign.h></stdalign.h>	<stdlib.h></stdlib.h>	<wctype.h></wctype.h>
<ctype.h></ctype.h>	<pre><limits.h></limits.h></pre>	<stdarg.h></stdarg.h>	<string.h></string.h>	
<errno.h></errno.h>	<locale.h></locale.h>	<stdbool.h></stdbool.h>	<tgmath.h></tgmath.h>	
<fenv.h></fenv.h>	<math.h></math.h>	<stddef.h></stddef.h>	<time.h></time.h>	
<float.h></float.h>	<setjmp.h></setjmp.h>	<stdint.h></stdint.h>	<uchar.h></uchar.h>	

- <sup>2</sup> Every C header, each of which has a name of the form name.h, behaves as if each name placed in the standard library namespace by the corresponding *cname* header is placed within the global namespace scope. It is unspecified whether these names are first declared or defined within namespace scope (3.3.6) of the namespace std and are then injected into the global namespace scope by explicit *using-declarations* (7.3.3).
- 3 [Example: The header <cstdlib> assuredly provides its declarations and definitions within the namespace std. It may also provide these names within the global namespace. The header <stdlib.h> assuredly provides the same declarations and definitions within the global namespace, much as in the C Standard. It may also provide these names within the namespace std. end example]

#### D.6 Old iostreams members

[depr.ios.members]

<sup>1</sup> The following member names are in addition to names specified in Clause 27:

§ D.6

```
namespace std {
  class ios_base {
  public:
    typedef T1 io_state;
    typedef T2 open_mode;
    typedef T3 seek_dir;
    typedef implementation-defined streamoff;
    typedef implementation-defined streampos;
    // remainder unchanged
  };
}
```

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.

- <sup>3</sup> 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.
- <sup>4</sup> 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.
- <sup>5</sup> The type streamoff is an implementation-defined type that satisfies the requirements of off\_type in 27.2.2.
- <sup>6</sup> The type streampos is an implementation-defined type that satisfies the requirements of pos\_type in 27.2.2.
- An implementation may provide the following additional member function, which has the effect of calling sbumpc() (27.6.3.2.3):

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
  class basic_streambuf {
  public:
    void stossc();
    // remainder unchanged
  };
}
```

8 An implementation may provide the following member functions that overload signatures specified in Clause 27:

```
namespace std {
  template<class charT, class traits> class basic_ios {
 public:
    void clear(io_state state);
    void setstate(io_state state);
    void exceptions(io_state);
    // remainder unchanged
  };
 class ios_base {
  public:
    // remainder unchanged
  };
  template<class charT, class traits = char_traits<charT> >
  class basic streambuf {
  public:
    pos_type pubseekoff(off_type off, ios_base::seek_dir way,
              ios_base::open_mode which = ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
              ios_base::open_mode which);
    // remainder unchanged
```

§ D.6

```
};
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> {
  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> {
  void open(const char* s, ios_base::open_mode mode);
  // remainder unchanged
};
```

<sup>9</sup> The effects of these functions is to call the corresponding member function specified in Clause 27.

#### D.7 char\* streams

[depr.str.strstreams]

The header **<strstream>** defines three types that associate stream buffers with character array objects and assist reading and writing such objects.

#### D.7.1 Class strstreambuf

[depr.strstreambuf]

```
namespace std {
  class strstreambuf : public basic_streambuf<char> {
  public:
    explicit strstreambuf(streamsize alsize_arg = 0);
    strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
    strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
    strstreambuf(const char* gnext_arg, streamsize n);
    strstreambuf(signed char* gnext_arg, streamsize n,
                 signed char* pbeg_arg = 0);
    strstreambuf(const signed char* gnext_arg, streamsize n);
    strstreambuf(unsigned char* gnext_arg, streamsize n,
                 unsigned char* pbeg_arg = 0);
    strstreambuf(const unsigned char* gnext_arg, streamsize n);
    virtual ~strstreambuf();
    void freeze(bool freezefl = true);
    char* str();
    int
         pcount();
  protected:
    virtual int_type overflow (int_type c = EOF);
    virtual int_type pbackfail(int_type c = EOF);
```

§ D.7.1

```
virtual int type underflow();
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                              ios_base::openmode which
                                = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp, ios_base::openmode which
                                = ios_base::in | ios_base::out);
    virtual streambuf* setbuf(char* s, streamsize n);
  private:
                                       // exposition only
    typedef T1 strstate;
                                       // exposition only
    static const strstate allocated;
    static const strstate constant;
                                       // exposition only
                                       // exposition only
    static const strstate dynamic;
                                       // exposition only
    static const strstate frozen;
                                       // exposition only
    strstate strmode;
                                       // exposition only
    streamsize alsize;
                                       // exposition only
    void* (*palloc)(size_t);
    void (*pfree)(void*);
                                       // exposition only
  };
}
```

- 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]
- <sup>3</sup> [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

1

Each object of class strstreambuf has a *seekable area*, delimited by the pointers seeklow and seekhigh. If gnext is a null pointer, the seekable area is undefined. Otherwise, seeklow equals gbeg and seekhigh is either pend, if pend is not a null pointer, or gend.

# D.7.1.1 strstreambuf constructors

[depr.strstreambuf.cons]

explicit strstreambuf(streamsize alsize\_arg = 0);

Effects: Constructs an object of class strstreambuf, initializing the base class with streambuf(). The postconditions of this function are indicated in Table 155.

Table 155 — strstreambuf(streamsize) effects

Element	Value
strmode	dynamic
alsize	alsize_arg
palloc	a null pointer
pfree	a null pointer

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 156.

Table 156 — strstreambuf(void\* (\*)(size\_t), void (\*)(void\*)) effects

Element	Value
strmode	dynamic
alsize	an unspecified value
palloc	palloc_arg
pfree	pfree_arg

Effects: Constructs an object of class strstreambuf, initializing the base class with streambuf(). The postconditions of this function are indicated in Table 157.

Table 157 — strstreambuf(charT\*, streamsize, charT\*) effects

Element	Value
strmode	0
alsize	an unspecified value
palloc	a null pointer
pfree	a null pointer

- gnext\_arg shall point to the first element of an array object whose number of elements N is determined as follows:
  - If n > 0, N is n.
  - If n == 0, N is std::strlen(gnext\_arg).
  - If n < 0, N is INT\_MAX. $^{337}$

337) The function signature strlen(const char\*) is declared in <cstring>. (21.7). The macro INT\_MAX is defined in <climits> (18.3).

```
5
        If pbeg_arg is a null pointer, the function executes:
        setg(gnext_arg, gnext_arg + N);
6
        Otherwise, the function executes:
          setg(gnext_arg, gnext_arg, pbeg_arg);
          setp(pbeg_arg, pbeg_arg + N);
          strstreambuf(const char* gnext_arg, streamsize n);
          strstreambuf(const signed char* gnext_arg, streamsize n);
          strstreambuf(const unsigned char* gnext_arg, streamsize n);
7
        Effects: Behaves the same as strstreambuf((char*)gnext_arg,n), except that the constructor also
        sets constant in strmode.
   virtual ~strstreambuf();
8
        Effects: Destroys an object of class strstreambuf. The function frees the dynamically allocated array
        object only if strmode & allocated != 0 and strmode & frozen == 0. (D.7.1.3 describes how a
        dynamically allocated array object is freed.)
   D.7.1.2 Member functions
                                                                          [depr.strstreambuf.members]
   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.
   char* str();
2
        Effects: Calls freeze(), then returns the beginning pointer for the input sequence, gbeg.
3
        Remarks: The return value can be a null pointer.
   int pcount() const;
4
        Effects: If the next pointer for the output sequence, pnext, is a null pointer, returns zero. Otherwise,
        returns the current effective length of the array object as the next pointer minus the beginning pointer
        for the output sequence, pnext - pbeg.
   D.7.1.3 strstreambuf overridden virtual functions
                                                                            [depr.strstreambuf.virtuals]
   int_type overflow(int_type c = EOF);
1
        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++.
2
             Returns (unsigned char)c.
          — If c == EOF, there is no character to append.
3
             Returns a value other than EOF.
```

- 4 Returns EOF to indicate failure.
- 5 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 (reallocate 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.
- 11 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();

15

- 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.
- 17 Returns EOF to indicate failure.
- 18 Remarks: The function can alter the number of read positions available as a result of any call.

 $338) \ \mathrm{An}$  implementation should consider  $\mathtt{alsize}$  in making this decision.

pos\_type seekoff(off\_type off, seekdir way, openmode which = in | out);

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 158.

Table 158 — seekoff position	ing
------------------------------	-----

Conditions	Result
(which & ios::in) != 0	positions the input sequence
(which & ios::out) != 0	positions the output sequence
(which & (ios::in	positions both the input and the output sequences
ios::out)) == (ios::in	
ios::out)) and	
way == either	
ios::beg or	
ios::end	
Otherwise	the positioning operation fails.

For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines newoff as indicated in Table 159.

Table 159 — newoff values

Condition	newoff Value
way == ios::beg	0
way == ios::cur	the next pointer minus the begin-
	ning pointer (xnext - xbeg).
way == ios::end	seekhigh minus the beginning
	pointer (seekhigh - xbeg).
If (newoff + off) <	the positioning operation fails
(seeklow - xbeg),	
or (seekhigh - xbeg) <	
(newoff + off)	

- 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)).

- 23 Effects: Alters the stream position within one of the controlled sequences, if possible, to correspond to the stream position stored in sp (as described below).
  - If (which & ios::in) != 0, positions the input sequence.
  - If (which & ios::out) != 0, positions the output sequence.

— If the function positions neither sequence, the positioning operation fails.

For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines newoff from sp.offset():

- If newoff is an invalid stream position, has a negative value, or has a value greater than (seekhigh seeklow), the positioning operation fails
- Otherwise, the function adds newoff to the beginning pointer xbeg and stores the result in the next pointer xnext.

Returns: pos\_type(newoff), constructed from the resultant offset newoff (of type off\_type), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is pos\_type(off\_type(-1)).

```
streambuf<char>* setbuf(char* s, streamsize n);
```

Effects: Implementation defined, except that setbuf(0, 0) has no effect.

#### D.7.2 Class istrstream

24

26

1

[depr.istrstream]

```
namespace std {
  class istrstream : public basic_istream<char> {
  public:
     explicit istrstream(const char* s);
     explicit istrstream(char* s);
     istrstream(const char* s, streamsize n);
     istrstream(char* s, streamsize n);
     virtual ~istrstream();

     strstreambuf* rdbuf() const;
     char *str();
     private:
        strstreambuf sb; // exposition only
     };
}
```

- The class istrstream supports the reading of objects of class strstreambuf. It supplies a strstreambuf object to control the associated array object. For the sake of exposition, the maintained data is presented here as:
  - sb, the strstreambuf object.

#### D.7.2.1 istrstream constructors

[depr.istrstream.cons]

```
explicit istrstream(const char* s);
explicit istrstream(char* s);
```

Effects: Constructs an object of class istrstream, initializing the base class with istream(&sb) and initializing sb with strstreambuf(s,0)). s shall designate the first element of an NTBS.

```
istrstream(const char* s, streamsize n);
```

2 Effects: Constructs an object of class istrstream, initializing the base class with istream(&sb) and initializing sb with strstreambuf(s,n)). s shall designate the first element of an array whose length is n elements, and n shall be greater than zero.

§ D.7.2.1

# D.7.2.2 Member functions

[depr.istrstream.members]

```
strstreambuf* rdbuf() const;

Returns: const_cast<strstreambuf*>(&sb).

char* str();

Returns: rdbuf()->str().
```

# D.7.3 Class ostrstream

[depr.ostrstream]

```
namespace std {
  class ostrstream : public basic_ostream<char> {
  public:
    ostrstream();
    ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
    virtual ~ostrstream();

    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
  private:
    strstreambuf sb; // exposition only
  };
}
```

- The class ostrstream supports the writing of objects of class strstreambuf. It supplies a strstreambuf object to control the associated array object. For the sake of exposition, the maintained data is presented here as:
  - sb, the strstreambuf object.

# D.7.3.1 ostrstream constructors

[depr.ostrstream.cons]

ostrstream();

1

Effects: Constructs an object of class ostrstream, initializing the base class with ostream(&sb) and initializing sb with strstreambuf()).

```
ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
```

- 2 Effects: Constructs an object of class ostrstream, initializing the base class with ostream(&sb), and initializing sb with one of 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)).<sup>339</sup>

§ D.7.3.1

<sup>339)</sup> The function signature  $strlen(const\ char*)$  is declared in  $\langle cstring \rangle$  (21.7).

```
D.7.3.2 Member functions
```

[depr.ostrstream.members]

```
strstreambuf* rdbuf() const;
1
        Returns: (strstreambuf*)&sb .
   void freeze(bool freezefl = true);
        Effects: Calls rdbuf()->freeze(freezef1).
   char* str();
3
        Returns: rdbuf()->str().
   int pcount() const;
        Returns: rdbuf()->pcount().
   D.7.4
          Class strstream
                                                                                     [depr.strstream]
    namespace std {
       class strstream
         : public basic_iostream<char> {
      public:
         // Types
         typedef char
                                                     char_type;
         typedef typename char_traits<char>::int_type int_type;
         typedef typename char_traits<char>::pos_type pos_type;
         typedef typename char_traits<char>::off_type off_type;
         // constructors/destructor
         strstream();
         strstream(char* s, int n,
                   ios_base::openmode mode = ios_base::in|ios_base::out);
         virtual ~strstream();
         // Members:
         strstreambuf* rdbuf() const;
         void freeze(bool freezefl = true);
         int pcount() const;
         char* str();
      private:
       strstreambuf sb; // exposition only
```

The class strstream supports reading and writing from objects of classs strstreambuf. It supplies a strstreambuf object to control the associated array object. For the sake of exposition, the maintained data is presented here as

— sb, the strstreambuf object.

§ D.7.4

[depr.strstream.cons]

D.7.4.1 strstream constructors

```
strstream();
1
        Effects: Constructs an object of class strstream, initializing the base class with iostream(&sb).
   strstream(char* s, int n,
             ios_base::openmode mode = ios_base::in|ios_base::out);
        Effects: Constructs an object of class strstream, initializing the base class with iostream(&sb) and
        initializing sb with one of the two constructors:
          — If (mode & app) == 0, then s shall designate the first element of an array of n elements. The
             constructor is strstreambuf(s,n,s).
          — If (mode & app) != 0, then s shall designate the first element of an array of n elements that
             contains an NTBS whose first element is designated by s. The constructor is strstreambuf(s,n,s
             + std::strlen(s)).
   D.7.4.2 strstream destructor
                                                                                   [depr.strstream.dest]
   virtual ~strstream()
1
        Effects: Destroys an object of class strstream.
   strstreambuf* rdbuf() const;
        Returns: &sb.
   D.7.4.3 strstream operations
                                                                                  [depr.strstream.oper]
   void freeze(bool freezefl = true);
        Effects: Calls rdbuf()->freeze(freezef1).
   char* str();
        Returns: rdbuf()->str().
   int pcount() const;
        Returns: rdbuf()->pcount().
   \mathbf{D.8}
         Function objects
                                                                               [depr.function.objects]
   D.8.1
           \mathbf{Base}
                                                                                             [depr.base]
  The class templates unary_function and binary_function are deprecated. A program shall not declare
   specializations of these templates.
     namespace std {
       template <class Arg, class Result>
       struct unary_function {
         typedef Arg
                        argument_type;
         typedef Result result_type;
       };
     }
   § D.8.1
                                                                                                     1213
```

```
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;
      };
    }
           Function adaptors
   D.8.2
                                                                                     [depr.adaptors]
  The adaptors ptr_fun, mem_fun, mem_fun_ref, and their corresponding return types are deprecated.
   [Note: The function template bind 20.8.9.1 provides a better solution. — end note]
   D.8.2.1 Adaptors for pointers to functions
                                                                    [depr.function.pointer.adaptors]
  To allow pointers to (unary and binary) functions to work with function adaptors the library provides:
   template <class Arg, class Result>
   class pointer_to_unary_function : public unary_function<Arg, Result> {
     explicit pointer_to_unary_function(Result (*f)(Arg));
    Result operator()(Arg x) const;
  };
2
        operator() returns f(x).
   template <class Arg, class Result>
    pointer_to_unary_function<Arg, Result> ptr_fun(Result (*f)(Arg));
3
        Returns: pointer_to_unary_function<Arg, Result>(f).
   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));
5
        Returns: pointer_to_binary_function<Arg1,Arg2,Result>(f).
6
        [Example:
          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]
```

§ D.8.2.1

# D.8.2.2 Adaptors for pointers to members

# [depr.member.pointer.adaptors]

The purpose of the following is to provide the same facilities for pointer to members as those provided for pointers to functions in D.8.2.1.

```
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;
  };
2
        mem_fun_t calls the member function it is initialized with given a pointer argument.
   template <class S, class T, class A> class mem_fun1_t
         : public binary_function<T*, A, S> {
   public:
     explicit mem_fun1_t(S (T::*p)(A));
     S operator()(T* p, A x) const;
   };
3
        mem_fun1_t calls the member function it is initialized with given a pointer argument and an additional
        argument of the appropriate type.
   template<class S, class T> mem_fun_t<S,T>
      mem_fun(S (T::*f)());
   template<class S, class T, class A> mem_fun1_t<S,T,A>
      mem_fun(S (T::*f)(A));
        mem_fun(&X::f) returns an object through which X::f can be called given a pointer to an X followed
        by the argument required for f (if any).
   template <class S, class T> class mem_fun_ref_t
         : public unary_function<T, S> {
   public:
     explicit mem_fun_ref_t(S (T::*p)());
     S operator()(T& p) const;
   };
        mem_fun_ref_t calls the member function it is initialized with given a reference argument.
   template <class S, class T, class A> class mem_fun1_ref_t
         : public binary_function<T, A, S> {
   public:
     explicit mem_fun1_ref_t(S (T::*p)(A));
     S operator()(T& p, A x) const;
   };
6
        mem_fun1_ref_t calls the member function it is initialized with given a reference argument and an
        additional argument of the appropriate type.
```

§ D.8.2.2

```
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));
7
         mem fun ref(&X::f) returns an object through which X::f can be called given a reference to an X
         followed by the argument required for f (if any).
   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;
   };
8
         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;
   };
9
         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);
10
         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> {
      explicit const_mem_fun_ref_t(S (T::*p)() const);
      S operator()(const T& p) const;
   };
11
         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> {
     explicit const_mem_fun1_ref_t(S (T::*p)(A) const);
     S operator()(const T& p, A x) const;
   };
```

§ D.8.2.2

const\_mem\_fun1\_ref\_t calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

```
template<class S, class T> const_mem_fun_ref_t<S,T>
    mem_fun_ref(S (T::*f)() const);
template<class S, class T, class A> const_mem_fun1_ref_t<S,T,A>
    mem_fun_ref(S (T::*f)(A) const);
```

mem\_fun\_ref(&X::f) returns an object through which X::f can be called given a reference to an X followed by the argument required for f (if any).

D.9 Binders [depr.lib.binders]

The binder1st, bind1st, binder2nd, and bind2nd are deprecated. [Note: The function template bind (20.8.9) provides a better solution. — end note]

# D.9.1 Class template binder1st

[depr.lib.binder.1st]

```
template <class Fn>
class binder1st
  : public unary_function<typename Fn::second_argument_type,
                          typename Fn::result_type> {
protected:
  Fn
  typename Fn::first_argument_type value;
public:
 binder1st(const Fn& x,
            const typename Fn::first_argument_type& y);
  typename Fn::result_type
    operator()(const typename Fn::second_argument_type& x) const;
 typename Fn::result_type
    operator()(typename Fn::second_argument_type& x) const;
};
   The constructor initializes op with x and value with y.
   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);
    Returns: binder1st<Fn>(fn, typename Fn::first_argument_type(x)).
```

## D.9.3 Class template binder2nd

1

2

[depr.lib.binder.2nd]

§ D.9.3

```
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)).
2
        [Example:
          find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5));
        finds the first integer in vector v greater than 5;
          find_if(v.begin(), v.end(), bind1st(greater<int>(), 5));
        finds the first integer in v less than 5. — end example]
```

D.10 auto\_ptr

[depr.auto.ptr]

The class template  $\mathtt{auto\_ptr}$  is deprecated. [Note: The class template  $\mathtt{unique\_ptr}$  (20.7.1) provides a better solution. —end note]

# D.10.1 Class template auto\_ptr

[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).
- <sup>2</sup> 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.

```
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();
```

§ D.10.1

```
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>
    public:
      typedef void element_type;
The class template auto_ptr provides a semantics of strict ownership. An auto_ptr owns the object it holds
a pointer to. Copying an auto_ptr copies the pointer and transfers ownership to the destination. If more than
one auto_ptr owns the same object at the same time the behavior of the program is undefined. [Note: The
uses of auto_ptr include providing temporary exception-safety for dynamically allocated memory, passing
ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from
a function. Instances of auto_ptr meet the requirements of MoveConstructible and MoveAssignable, but
do not meet the requirements of CopyConstructible and CopyAssignable. — end note
D.10.1.1 auto_ptr constructors
                                                                                        [auto.ptr.cons]
explicit auto_ptr(X* p =0) throw();
      Postconditions: *this holds the pointer p.
auto_ptr(auto_ptr& a) throw();
      Effects: Calls a.release().
      Postconditions: *this holds the pointer returned from a.release().
template<class Y> auto_ptr(auto_ptr<Y>& a) throw();
      Requires: Y* can be implicitly converted to X*.
      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.
```

1219

1

2

3

4

5 6

7

8

9

10

11

12

§ D.10.1.1

```
~auto_ptr() throw();
13
         Requires: The expression delete get() is well formed.
14
         Effects: delete get().
   D.10.1.2 auto_ptr members
                                                                                     [auto.ptr.members]
   X& operator*() const throw();
1
         Requires: get() != 0
2
         Returns: *get()
   X* operator->() const throw();
         Returns: get()
   X* get() const throw();
         Returns: The pointer *this holds.
   X* release() throw();
5
         Returns: get()
6
         Postcondition: *this holds the null pointer.
   void reset(X* p=0) throw();
7
         Effects: If get() != p then delete get().
8
         Postconditions: *this holds the pointer p.
   D.10.1.3 auto_ptr conversions
                                                                                          [auto.ptr.conv]
   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().
   template<class Y> operator auto_ptr_ref<Y>() throw();
3
         Returns: An auto_ptr_ref<Y> that holds *this.
   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().
   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.
         Returns: *this
   § D.10.1.3
                                                                                                      1220
```

# D.11 Violating exception-specifications

# [exception.unexpected]

# D.11.1 Type unexpected\_handler

[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.

- 2 Required behavior: An unexpected\_handler shall not return. See also 15.5.2.
- 3 Default behavior: The implementation's default unexpected\_handler calls std::terminate().

# D.11.2 set\_unexpected

1

1

[set.unexpected]

unexpected\_handler set\_unexpected(unexpected\_handler f) noexcept;

- Effects: Establishes the function designated by f as the current unexpected\_handler.
- 2 Remark: It is unspecified whether a null pointer value designates the default unexpected\_handler.
- 3 Returns: The previous unexpected\_handler.

# D.11.3 get\_unexpected

[get.unexpected]

unexpected\_handler get\_unexpected() noexcept;

Returns: The current unexpected\_handler. [Note: This may be a null pointer value. —end note]

# D.11.4 unexpected

[unexpected]

[[noreturn]] void unexpected();

- Remarks: Called by the implementation when a function exits via an exception not allowed by its exception-specification (15.5.2), in effect after evaluating the throw-expression (D.11.1). May also be called directly by the program.
- 2 Effects: Calls the current unexpected\_handler function. [Note: A default unexpected\_handler is always considered a callable handler in this context. —end note]

§ D.11.4

# Annex E (normative) Universal character names for identifier characters [charname]

# E.1 Ranges of characters allowed

[charname.allowed]

```
00A8, 00AA, 00AD, 00AF, 00B2-00B5, 00B7-00BA, 00BC-00BE, 00C0-00D6, 00D8-00F6, 00F8-00FF
0100-167F, 1681-180D, 180F-1FFF
200B-200D, 202A-202E, 203F-2040, 2054, 2060-206F
2070-218F, 2460-24FF, 2776-2793, 2C00-2DFF, 2E80-2FFF
3004-3007, 3021-302F, 3031-303F
3040-D7FF
F900-FD3D, FD40-FDCF, FDF0-FE44, FE47-FFFD
10000-1FFFD, 20000-2FFFD, 30000-3FFFD, 40000-4FFFD, 50000-5FFFD, 60000-6FFFD, 70000-7FFFD, 80000-8FFFD, 90000-9FFFD, A0000-AFFFD, B0000-BFFFD, C0000-CFFFD, D0000-DFFFD, E0000-EFFFD
```

# E.2 Ranges of characters disallowed initially

[charname.disallowed]

 ${\tt 0300-036F,\ 1DC0-1DFF,\ 20D0-20FF,\ FE20-FE2F}$ 

§ E.2 1222

# Annex F (informative) Cross references

[xref]

This annex lists each section label and the corresponding section number, in alphabetical order by label. All of the section labels are the same as in the 2003 standard, except:

- labels that begin with lib. in the 2003 standard have had the lib. removed so that they do not all appear in the same part of this list. For example, in the 2003 standard, the non-modifying sequence algorithms were found in a section with the label [lib.alg.nonmodifying]. The label for that section is now [alg.nonmodifying].
- the label for Annex B has been changed from [limits] to [implimits]. The label [limits] refers to section 18.3.2.

$\mathbf{A}$	alg.reverse 25.3.10
accumulate 26.7.2	alg.rotate 25.3.11
adjacent.difference 26.7.5	alg.search 25.2.13
adjustfield.manip 27.5.6.2	alg.set.operations 25.4.5
alg.adjacent.find 25.2.8	alg.sort 25.4.1
alg.all_of 25.2.1	alg.sorting 25.4
alg.any_of 25.2.2	alg.swap 25.3.3
alg.binary.search 25.4.3	alg.transform 25.3.4
alg.c.library 25.5	alg.unique 25.3.9
alg.copy 25.3.1	algorithm.stable 17.6.5.7
alg.count 25.2.9	algorithms 25
alg.equal 25.2.11	algorithms.general 25.1
alg.fill 25.3.6	alloc.errors 18.6.2
alg.find 25.2.5	allocator.adaptor 20.12
alg.find.end 25.2.6	allocator.adaptor.cnstr 20.12.3
alg.find.first.of 25.2.7	allocator.adaptor.members 20.12.4
alg.foreach 25.2.4	allocator.adaptor.syn 20.12.1
alg.generate 25.3.7	allocator.adaptor.types 20.12.2
alg.heap.operations 25.4.6	allocator.globals 20.6.9.2
alg.is_permutation 25.2.12	allocator.members 20.6.9.1
alg.lex.comparison 25.4.8	allocator.requirements 17.6.3.5
alg.merge 25.4.4	allocator.tag 20.6.6
alg.min.max 25.4.7	allocator.traits 20.6.8
alg.modifying.operations 25.3	allocator.traits.members 20.6.8.2
alg.move 25.3.2	allocator.traits.types 20.6.8.1
alg.none_of 25.2.3	allocator.uses 20.6.7
alg.nonmodifying 25.2	allocator.uses.construction 20.6.7.2
alg.nth.element 25.4.2	allocator.uses.trait 20.6.7.1
alg.partitions 25.3.13	alt.headers 17.6.4.4
alg.permutation.generators 25.4.9	arithmetic.operations 20.8.4
alg.random.shuffle 25.3.12	array 23.3.2
alg.remove 25.3.8	array.cons 23.3.2.2
alg.replace 25.3.5	array.data 23.3.2.5

au	
array.fill 23.3.2.6	basic.fundamental 3.9.1
array.overview 23.3.2.1	basic.funscope 3.3.5
array.size 23.3.2.4	basic.ios.cons 27.5.5.2
array.special 23.3.2.3	basic.ios.members 27.5.5.3
array.swap 23.3.2.7	basic.life 3.8
array.tuple 23.3.2.9	basic.link 3.5
array.zero 23.3.2.8	basic.lookup 3.4
assertions 19.3	basic.lookup.argdep 3.4.2
associative 23.4	basic.lookup.classref 3.4.5
associative.general 23.4.1	basic.lookup.elab 3.4.4
associative.map.syn 23.4.2	basic.lookup.qual 3.4.3
associative.reqmts 23.2.4	basic.lookup.udir 3.4.6
associative.reqmts.except 23.2.4.1	basic.lookup.unqual 3.4.1
associative.set.syn 23.4.3	basic.lval 3.10
atomics 29	basic.namespace 7.3
atomics.fences 29.8	basic.scope 3.3
atomics.flag 29.7	basic.scope.class 3.3.7
atomics.general 29.1	basic.scope.declarative 3.3.1
atomics.lockfree 29.4	basic.scope.enum 3.3.8
atomics.order 29.3	basic.scope.hiding 3.3.10
atomics.syn 29.2	basic.scope.local 3.3.3
atomics.types.generic 29.5	basic.scope.namespace 3.3.6
atomics.types.operations 29.6	basic.scope.pdecl 3.3.2
atomics.types.operations.arith 29.6.3	basic.scope.proto 3.3.4
atomics.types.operations.general 29.6.1	basic.scope.temp 3.3.9
atomics.types.operations.pointer 29.6.4	basic.start 3.6
atomics.types.operations.req 29.6.5	basic.start.init 3.6.2
atomics.types.operations.templ 29.6.2	basic.start.main 3.6.1
auto.ptr D.10.1	basic.start.term 3.6.3
auto.ptr.cons D.10.1.1	basic.stc 3.7
auto.ptr.conv D.10.1.3	basic.stc.auto 3.7.3
auto.ptr.members D.10.1.2	basic.stc.dynamic 3.7.4
•	basic.stc.dynamic.allocation 3.7.4.1
В	basic.stc.dynamic.deallocation 3.7.4.2
back.insert.iter.cons 24.5.2.2.1	basic.stc.dynamic.safety 3.7.4.3
back.insert.iter.op* 24.5.2.2.3	basic.stc.inherit 3.7.5
back.insert.iter.op++ 24.5.2.2.4	basic.stc.static 3.7.1
back.insert.iter.op= 24.5.2.2.2	basic.stc.thread 3.7.2
back.insert.iter.ops 24.5.2.2	basic.string 21.4
back.insert.iterator 24.5.2.1	basic.string.hash 21.6
back.inserter 24.5.2.2.5	basic.type.qualifier 3.9.3
bad.alloc 18.6.2.1	basic.types 3.9
bad.cast 18.7.2	bidirectional.iterators 24.2.6
bad.exception 18.8.2	binary.search 25.4.3.4
bad.typeid 18.7.3	bind 20.8.9
basefield.manip 27.5.6.3	bitmask.types 17.5.2.1.3
basic 3	bitset.cons 20.5.1
basic.align 3.11	bitset.hash 20.5.3
basic.compound 3.9.2	bitset.members 20.5.2
basic.def 3.1	bitset.operators 20.5.4
basic.def 3.1 basic.def.odr 3.2	bitwise operations 20.8.7
Santa Carolia Car	20.011

C c.files 27.9.2 c.limits 18.3.3 c.locales 22.6 c.malloc 20.6.13 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 ccmplx 26.4.10 cfenv 26.3 cfenv.syn 26.3.1 char.traits 21.2	class.gslice.overview 26.6.6.1 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 11.6 class.protected 11.4 class.qual 3.4.3.1 class.slice 26.6.4 class.slice.overview 26.6.4.1 class.static 9.4 class.static.data 9.4.2
char.traits.require 21.2.1	class.static.mfct 9.4.1
char.traits.specializations 21.2.3	class.temporary 12.2
char.traits.specializations.char 21.2.3.1	class.this 9.3.2
char.traits.specializations.char16_t 21.2.3.2	class.union 9.5
char.traits.specializations.char32_t 21.2.3.3	class.virtual 10.3
char.traits.specializations.wchar.t 21.2.3.4	classification 22.3.3.1
char.traits.typedefs 21.2.2	cmplx.over $26.4.9$
character.seq 17.5.2.1.4	comparisons 20.8.5
charname E	complex 26.4.2
charname.allowed E.1	complex.member.ops 26.4.5
charname.disallowed E.2	complex.members 26.4.4
class 9	complex.numbers 26.4
class.abstract 10.4	complex.ops 26.4.6
class.access 11	complex.special 26.4.3
class.access.base 11.2	complex.syn 26.4.1
class.access.nest 11.7	complex.transcendentals 26.4.8
class.access.spec 11.1	complex.value.ops 26.4.7
class.access.virt 11.5	compliance 17.6.1.3
class.base.init 12.6.2	conforming 17.6.5
class.bit 9.6 class.cdtor 12.7	conforming.overview 17.6.5.1 cons.slice 26.6.4.2
class.conv 12.7	constexpr.functions 17.6.5.6
class.conv.ctor 12.3.1	constraints 17.6.4
class.conv.fct 12.3.2	constraints 17.0.4 constraints.overview 17.6.4.1
class.copy 12.8	container.adaptors 23.6
class.ctor 12.1	container.adaptors 25.0 container.adaptors.general 23.6.1
class.derived 10	container.requirements 23.2
class.dtor 12.4	container.requirements.dataraces 23.2.2
class.expl.init 12.6.1	container.requirements.general 23.2.1
class.free 12.5	containers 23
class.friend 11.3	containers general 23.1
class.gslice 26.6.6	contents 17.6.1.1
<u> </u>	

conv 4	dcl.fct 8.3.5
conv.array 4.2	dcl.fct.def 8.4
conv.bool 4.12	dcl.fct.def.default 8.4.2
conv.double 4.8	dcl.fct.def.delete 8.4.3
conv.fipint 4.9	dcl.fct.def.general 8.4.1
conv.fprom 4.6	dcl.fct.default 8.3.6
conv.func 4.3	dcl.fct.spec 7.1.2
	dcl.friend 7.1.4
conv.integral 4.7 conv.lval 4.1	del.init 8.5
conv.nvar 4.1	
	dcl.init.aggr 8.5.1 dcl.init.list 8.5.4
conv.prom 4.5	dcl.init.ref 8.5.3
conv.ptr 4.10	
conv.qual 4.4 conv.rank 4.13	dcl.init.string 8.5.2 dcl.link 7.5
conventions 17.5.2	dcl.meaning 8.3
conversions 22.3.3.2	dcl.mptr 8.3.3
conversions.buffer 22.3.3.2.3	dcl.name 8.1
conversions.character 22.3.3.2.1	dcl.ptr 8.3.1
conversions.string 22.3.3.2.2	dcl.ref 8.3.2
cpp 16	dcl.spec 7.1
cpp.concat 16.3.3	dcl.spec.auto 7.1.6.4
cpp.cond 16.1	del.stc 7.1.1
cpp.error 16.5	dcl.type 7.1.6
cpp.include 16.2	dcl.type.cv 7.1.6.1
cpp.line 16.4	dcl.type.elab 7.1.6.3
cpp.null 16.7	dcl.type.simple 7.1.6.2
cpp.pragma 16.6	dcl.typedef 7.1.3
cpp.pragma.op 16.9	declval 20.2.4
cpp.predefined 16.8	default.allocator 20.6.9
cpp.replace 16.3	definitions 17.3
cpp.rescan 16.3.4	defns.additional 17.4
cpp.scope 16.3.5	defns.arbitrary.stream
cpp.stringize 16.3.2	defns.argument
cpp.subst 16.3.1	defns.argument.macro
cstdint 18.4	defns.argument.templ
cstdint.syn 18.4.1	defns.argument.throw
	defns.block
D	defns.blocked
date.time 20.11.8	defns.character
dcl.align 7.6.2	defns.character.container
dcl.ambig.res 8.2	defns.comparison
dcl.array 8.3.4	defns.component
dcl.asm 7.4	defns.cond.supp
dcl.attr 7.6	defns.deadlock
dcl.attr.depend 7.6.4	defns.default.behavior.func
dcl.attr.grammar 7.6.1	defns.default.behavior.impl
dcl.attr.noreturn 7.6.3	defns.diagnostic
dcl.constexpr 7.1.5	defns.dynamic.type
del.del 7	defns.dynamic.type.prvalue
dcl.decl 8	defns.handler
dcl.enum 7.2	defns.ill.formed

	D = 0
defins.impl.defined	depr.istrstream D.7.2
defns.impl.limits	depr.istrstream.cons D.7.2.1
defns.iostream.templates	depr.istrstream.members D.7.2.2
defns.locale.specific	depr.lib.bind.1st D.9.2
defns.modifier	depr.lib.bind.2nd D.9.4
defns.move.assign	depr.lib.binder.1st D.9.1
defns.move.constr	depr.lib.binder.2nd D.9.3
defns.multibyte	depr.lib.binders D.9
defns.ntcts	depr.member.pointer.adaptors D.8.2.2
defns.obj.state	depr.ostrstream D.7.3
defns.observer	depr.ostrstream.cons D.7.3.1
defns.parameter	depr.ostrstream.members D.7.3.2
defns.parameter.macro	depr.register D.2
defns.parameter.templ	depr.str.strstreams D.7
defns.regex.collating.element	depr.strstream D.7.4
defns.regex.finite.state.machine	depr.strstream.cons D.7.4.1
defns.regex.format.specifier	depr.strstream.dest D.7.4.2
defns.regex.matched	depr.strstream.oper D.7.4.3
defns.regex.primary.equivalence.class	depr.strstreambuf D.7.1
defns.regex.regular.expression	depr.strstreambuf.cons D.7.1.1
defns.regex.subexpression	depr.strstreambuf.members $D.7.1.2$
defns.replacement	depr.strstreambuf.virtuals D.7.1.3
defns.repositional.stream	deque 23.3.3
defns.required.behavior	deque.capacity 23.3.3.3
defns.reserved.function	deque.cons 23.3.3.2
defns.signature	deque.modifiers 23.3.3.4
defns.signature.member	deque.overview 23.3.3.1
defns.signature.member.spec	deque.special 23.3.3.5
defns.signature.member.templ	derivation 17.6.5.11
defns.signature.spec	derived.classes 17.6.4.5
defns.signature.templ	description 17.5
defns.stable	diagnostics 19
defns.static.type	diagnostics.general 19.1
defns.traits	diff C
defns.unblock	diff.basic C.1.2
defns.undefined	diff.char16 C.3.2.1
defns.unspecified	diff.class C.1.8
defns.valid	diff.conv C.1.3
defns.well.formed	diff.cpp C.1.10
denorm.style 18.3.2.6	diff.cpp03 $C.2$
depr D	diff.cpp03.algorithms C.2.13
depr.adaptors D.8.2	diff.cpp03.containers C.2.12
depr.auto.ptr D.10	diff.cpp03.dcl.dcl C.2.3
depr.base D.8.1	diff.cpp03.dcl.decl C.2.4
depr.c.headers D.5	diff.cpp03.diagnostics C.2.9
depr.except.spec D.4	diff.cpp03.expr $C.2.2$
depr.function.objects D.8	diff.cpp03.input.output C.2.15
depr.function.pointer.adaptors D.8.2.1	diff.cpp03.language.support C.2.8
depr.impldec D.3	diff.cpp03.lex C.2.1
depr.incr.bool D.1	diff.cpp03.library C.2.7
depr.ios.members D.6	diff.cpp03.numerics C.2.14
•	**

diff.cpp03.special C.2.5	expr.const.cast 5.2.11
diff.cpp03.strings C.2.11	expr.delete 5.3.5
diff.cpp03.temp C.2.6	expr.dynamic.cast 5.2.7
diff.cpp03.utilities C.2.10	expr.eq $5.10$
diff.dcl C.1.6	expr.log.and 5.14
diff.decl C.1.7	expr.log.or 5.15
diff.expr C.1.4	expr.mptr.oper 5.5
diff.header.iso646.h C.3.2.3	expr.mul 5.6
diff.iso C.1	expr.new 5.3.4
diff.lex C.1.1	expr.or 5.13
diff.library C.3	expr.post 5.2
diff.malloc C.3.4.2	expr.post.incr 5.2.6
diff.mods.to.behavior C.3.4	expr.pre.incr 5.3.2
diff.mods.to.declarations C.3.3	expr.prim 5.1
diff.mods.to.definitions C.3.2	expr.prim.general 5.1.1
diff.mods.to.headers C.3.1	expr.prim.lambda 5.1.2
diff.null C.3.2.4	expr.pseudo 5.2.4
diff.offsetof C.3.4.1	expr.ref 5.2.5
diff.special C.1.9	expr.reinterpret.cast 5.2.10
diff.stat C.1.5	expr.rel 5.9
diff.wchar.t C.3.2.2	expr.shift 5.8
domain.error 19.2.2	expr.sizeof 5.3.3
	expr.static.cast 5.2.9
${f E}$	expr.sub 5.2.1
enumerated.types 17.5.2.1.2	expr.type.conv 5.2.3
equal.range 25.4.3.3	expr.typeid 5.2.8
errno 19.4	expr.unary 5.3
error.reporting 27.5.6.5	expr.unary.noexcept 5.3.7
except 15	expr.unary.op 5.3.1
except.ctor 15.2	expr.xor 5.12
except.handle 15.3	ext.manip 27.7.5
except.nested 18.8.6	extern.names 17.6.4.3.3
except.spec 15.4	extern.types 17.6.4.3.4
except.special 15.5	extern.types 17.0.1.5.1
except.terminate 15.5.1	F
except.throw 15.1	facet.ctype.char.dtor 22.4.1.3.1
except.uncaught 15.5.3	facet.ctype.char.members 22.4.1.3.2
except.unexpected 15.5.2	facet.ctype.char.statics 22.4.1.3.3
exception 18.8.1	facet.ctype.char.virtuals 22.4.1.3.4
exception.terminate 18.8.3	facet.ctype.special 22.4.1.3
exception.unexpected D.11	facet.num.get.members 22.4.2.1.1
expr 5	facet.num.get.nichisers 22.4.2.1.1 facet.num.get.virtuals 22.4.2.1.2
expr.add 5.7	facet.num.put.members 22.4.2.1.2
expr.alignof 5.3.6	facet.num.put.inembers 22.4.2.2.1 facet.num.put.virtuals 22.4.2.2.2
expr.ass 5.17	facet.numpunct 22.4.3
expr.bit.and 5.11	facet.numpunct.members 22.4.3.1.1
expr.call 5.2.2	<del>-</del>
•	facet.numpunct.virtuals 22.4.3.1.2
expr.cast 5.4	facets.examples 22.4.8 file.streams 27.9
expr.comma 5.18	
expr.cond 5.16	filebuf 27.9.1.1
expr.const 5.19	filebuf.assign 27.9.1.3

61.1.6.07.0.1.0	C + 20.6
filebuf.cons 27.9.1.2	futures 30.6
filebuf.members 27.9.1.4	futures.async 30.6.8
filebuf.virtuals 27.9.1.5	futures.errors 30.6.2
floatfield.manip 27.5.6.4	futures.future_error 30.6.3
fmtflags.manip 27.5.6.1	futures.overview 30.6.1
fmtflags.state 27.5.3.2	futures.promise 30.6.5
forward 20.2.3	futures.shared_future 30.6.7
forward.iterators 24.2.5	futures.state 30.6.4
forwardlist 23.3.4	futures.task 30.6.9
forwardlist.access 23.3.4.4	futures.task.members 30.6.9.1
forwardlist.cons 23.3.4.2	futures.task.nonmembers 30.6.9.2
forwardlist.iter 23.3.4.3	futures.unique_future 30.6.6
forwardlist.modifiers 23.3.4.5	
forwardlist.ops 23.3.4.6	$\mathbf{G}$
forwardlist.overview 23.3.4.1	get.new.handler $18.6.2.5$
forwardlist.spec 23.3.4.7	get.terminate 18.8.3.3
fpos 27.5.4	get.unexpected D.11.3
fpos.members 27.5.4.1	global.functions 17.6.5.4
fpos.operations 27.5.4.2	global.names 17.6.4.3.2
front.insert.iter.cons 24.5.2.4.1	gram A
front.insert.iter.op* 24.5.2.4.3	gram.basic A.3
front.insert.iter.op 24.5.2.4.3 front.insert.iter.op++ 24.5.2.4.4	_
front.insert.iter.op= 24.5.2.4.4 front.insert.iter.op= 24.5.2.4.2	gram.class A.8
*	gram.cpp A.14
front.insert.iter.ops 24.5.2.4	gram.dcl A.6
front.insert.iterator 24.5.2.3	gram.decl A.7
front.inserter 24.5.2.4.5	gram.derived A.9
fstream 27.9.1.14	gram.except A.13
fstream.assign 27.9.1.16	gram.expr A.4
fstream.cons 27.9.1.15	gram.key A.1
fstream.members 27.9.1.17	gram.lex A.2
fstreams 27.9.1	gram.over A.11
func.bind 20.8.9.1	gram.special A.10
func.bind.bind 20.8.9.1.2	gram.stmt A.5
func.bind.isbind 20.8.9.1.1	gram.temp A.12
func.bind.place 20.8.9.1.3	gslice.access 26.6.6.3
func.def 20.8.1	gslice.array.assign 26.6.7.2
func.memfn 20.8.10	gslice.array.comp.assign 26.6.7.3
func.require 20.8.2	gslice.array.fill 26.6.7.4
func.wrap 20.8.11	gslice.cons 26.6.6.2
func.wrap.badcall 20.8.11.1	
func.wrap.badcall.const 20.8.11.1.1	Н
func.wrap.func 20.8.11.2	handler.functions 17.6.4.7
func.wrap.func.alg 20.8.11.2.7	hash.requirements 17.6.3.4
func.wrap.func.cap 20.8.11.2.3	headers 17.6.1.2
func.wrap.func.con 20.8.11.2.1	110000112
func.wrap.func.inv 20.8.11.2.4	I
func.wrap.func.mod 20.8.11.2.2	ifstream 27.9.1.6
func.wrap.func.nullptr 20.8.11.2.6	
func.wrap.func.targ 20.8.11.2.5	ifstream.assign 27.9.1.8 ifstream.cons 27.9.1.7
- ~	
function.objects 20.8	
functions.within.classes 17.5.2.2	implimits B

includes 25.4.5.1	iostream.limits.imbue 27.2.1
indirect.array.assign 26.6.9.2	iostream.objects 27.4
indirect.array.comp.assign 26.6.9.3	iostream.objects.overview 27.4.1
indirect.array.fill 26.6.9.4	iostreamclass 27.7.2.5
inner.product 26.7.3	iostreams.base 27.5
input.iterators 24.2.3	iostreams.base.overview 27.5.1
input.output 27	iostreams.limits.pos 27.2.2
input.output.general 27.1	iostreams.requirements 27.2
input.streams 27.7.2	iostreams.threadsafety 27.2.3
insert.iter.cons 24.5.2.6.1	is.heap 25.4.6.5
insert.iter.op* 24.5.2.6.3	is.sorted 25.4.1.5
insert.iter.op++ $24.5.2.6.4$	istream 27.7.2.1
insert.iter.op= $24.5.2.6.2$	istream.assign 27.7.2.1.2
insert.iter.ops 24.5.2.6	istream.cons 27.7.2.1.1
insert.iterator 24.5.2.5	istream.formatted 27.7.2.2
insert.iterators 24.5.2	istream.formatted.arithmetic 27.7.2.2.2
inserter 24.5.2.6.5	istream.formatted.reqmts 27.7.2.2.1
intro 1	istream.iterator 24.6.1
intro.ack 1.11	istream.iterator.cons 24.6.1.1
intro.compliance 1.4	istream.iterator.ops 24.6.1.2
intro.defs 1.3	istream.manip 27.7.2.4
intro.execution 1.9	istream.rvalue 27.7.2.6
intro.memory 1.7	istream.unformatted 27.7.2.3
intro.multithread 1.10	istream::extractors 27.7.2.3
intro.object 1.8	istream::sentry 27.7.2.1.3
intro.refs 1.2	istreamfuserity 27.7.2.1.3
intro.scope 1.1	istreambuf.iterator.cons 24.6.3.2
intro.scope 1.1 intro.structure 1.5	istreambuf.iterator::equal 24.6.3.5
invalid.argument 19.2.3	istreambuf.iterator::op!= 24.6.3.7
ios 27.5.5	istreambuf.iterator::op:= 24.6.3.7 istreambuf.iterator::op* 24.6.3.3
ios.base 27.5.3	
	istreambuf.iterator::op++ 24.6.3.4 istreambuf.iterator::op== 24.6.3.6
ios.base.callback 27.5.3.6	
ios.base.cons 27.5.3.7	istreambuf.iterator::proxy 24.6.3.1
ios.base.locales 27.5.3.3	istringstream 27.8.3
ios.base.storage 27.5.3.5	istringstream.assign 27.8.3.2
ios.members.static 27.5.3.4	istringstream.cons 27.8.3.1
ios.overview 27.5.5.1	istringstream.members 27.8.3.3
ios.types 27.5.3.1	iterator.basic 24.4.2
ios::failure 27.5.3.1.1	iterator.iterators 24.2.2
ios::fmtflags 27.5.3.1.2	iterator.operations 24.4.4
ios::Init 27.5.3.1.6	iterator.primitives 24.4
ios::iostate 27.5.3.1.3	iterator.range 24.6.5
ios::openmode 27.5.3.1.4	iterator.requirements 24.2
ios::seekdir 27.5.3.1.5	iterator.requirements.general 24.2.1
iostate.flags 27.5.5.4	iterator.synopsis 24.3
iostream.assign 27.7.2.5.3	iterator.traits 24.4.1
iostream.cons 27.7.2.5.1	iterators 24
iostream.dest 27.7.2.5.2	iterators.general 24.1
iostream.format 27.7	
iostream.format.overview 27.7.1	J
iostream.forward 27.3	K

L	locale.convenience 22.3.3
length.error 19.2.4	locale.ctype 22.4.1.1
lex 2	locale.ctype.byname 22.4.1.2
lex.bool 2.14.6	locale.ctype.members 22.4.1.1.1
lex.ccon 2.14.3	locale.ctype.virtuals 22.4.1.1.2
lex.charset 2.3	locale.facet 22.3.1.1.2
lex.comment 2.8	locale.global.templates 22.3.2
lex.digraph 2.6	locale.id 22.3.1.1.3
lex.ext 2.14.8	locale.members 22.3.1.3
lex.fcon 2.14.4	locale.messages 22.4.7.1
lex.header 2.9	locale.messages.byname 22.4.7.2
lex.icon 2.14.2	locale.messages.members 22.4.7.1.1
lex.key 2.12	locale.messages.virtuals 22.4.7.1.2
lex.literal 2.14	locale.money.get 22.4.6.1
lex.literal.kinds 2.14.1	locale.money.get.members 22.4.6.1.1
lex.name 2.11	locale.money.get.virtuals 22.4.6.1.2
lex.nullptr 2.14.7	locale.money.put 22.4.6.2
lex.operators 2.13	locale.money.put.members 22.4.6.2.1
lex.phases 2.2	locale.money.put.virtuals 22.4.6.2.2
lex.ppnumber 2.10	locale.moneypunct 22.4.6.3
lex.pptoken 2.5	locale.moneypunct.byname 22.4.6.4
lex.separate 2.1	locale.moneypunct.members 22.4.6.3.1
lex.string 2.14.5	locale.moneypunct.virtuals 22.4.6.3.2
lex.token 2.7	locale.nm.put 22.4.2.2
lex.trigraph 2.4	locale.num.get 22.4.2.1
lib.types.movedfrom 17.6.5.15	locale.numpunct 22.4.3.1
library 17	locale.numpunct.byname 22.4.3.2
library.c 17.2	locale.operators 22.3.1.4
library.general 17.1	locale.statics 22.3.1.5
limits 18.3.2	locale.stdcvt 22.5
limits.numeric 18.3.2.1	locale.syn 22.2
limits.syn 18.3.2.2	locale.time.get 22.4.5.1
list 23.3.5	locale.time.get.byname 22.4.5.2
list.capacity 23.3.5.3	locale.time.get.members 22.4.5.1.1
list.cons 23.3.5.2	locale.time.get.virtuals 22.4.5.1.2
list.modifiers 23.3.5.4	locale.time.put 22.4.5.3
list.ops 23.3.5.5	locale.time.put.byname 22.4.5.4
list.overview 23.3.5.1	locale.time.put.members 22.4.5.3.1
list.special 23.3.5.6	locale.time.put.virtuals 22.4.5.3.2
locale 22.3.1	locale.types 22.3.1.1
locale.categories 22.4	locales 22.3
locale.category 22.3.1.1.1	localization 22
locale.codecvt 22.4.1.4	localization.general 22.1
locale.codecvt.byname 22.4.1.5	logic.error 19.2.1
locale.codecvt.members 22.4.1.4.1	logical.operations 20.8.6
locale.codecvt.virtuals 22.4.1.4.2	lower.bound $25.4.3.1$
locale.collate 22.4.4.1	
locale.collate.byname 22.4.4.2	${f M}$
locale.collate.members 22.4.4.1.1	macro.names 17.6.4.3.1
locale.collate.virtuals 22.4.4.1.2	make.heap 25.4.6.3
locale.cons 22.3.1.2	map 23.4.4

00.4.4.0	1 20.45.2
map.access 23.4.4.3	multimap.cons 23.4.5.2
map.cons 23.4.4.2	multimap.modifiers 23.4.5.3
map.modifiers 23.4.4.4	multimap.ops 23.4.5.4
map.ops 23.4.4.5	multimap.overview 23.4.5.1
map.overview 23.4.4.1	multimap.special 23.4.5.5
map.special 23.4.4.6	multiset 23.4.7
mask.array.assign 26.6.8.2	multiset.cons 23.4.7.2
mask.array.comp.assign 26.6.8.3	multiset.overview 23.4.7.1
mask.array.fill 26.6.8.4	multiset.special 23.4.7.3
member.functions 17.6.5.5	
memory 20.6	N
memory.general 20.6.1	namespace.alias 7.3.2
memory.syn 20.6.2	namespace.constraints 17.6.4.2
meta 20.9	namespace.def 7.3.1
meta.help 20.9.3	namespace.memdef 7.3.1.2
meta.rel 20.9.6	namespace.posix 17.6.4.2.2
meta.rqmts 20.9.1	namespace.qual 3.4.3.2
meta.trans 20.9.7	namespace.std 17.6.4.2.1
meta.trans.arr 20.9.7.4	namespace.udecl 7.3.3
meta.trans.cv 20.9.7.1	namespace.udir 7.3.4
meta.trans.other 20.9.7.6	namespace.unnamed 7.3.1.1
meta.trans.ptr 20.9.7.5	narrow.stream.objects 27.4.2
meta.trans.ref 20.9.7.2	negators 20.8.8
meta.trans.sign 20.9.7.3	new.badlength 18.6.2.2
meta.type.synop 20.9.2	new.delete 18.6.1
meta.unary 20.9.4	new.delete.array 18.6.1.2
meta.unary.cat 20.9.4.1	new.delete.dataraces 18.6.1.4
meta.unary.comp 20.9.4.2	new.delete.placement 18.6.1.3
meta.unary.prop 20.9.4.3	new.delete.single 18.6.1.1
meta.unary.prop.query 20.9.5	new.handler 18.6.2.3
mismatch 25.2.10	nullablepointer.requirements 17.6.3.3
move.iter.nonmember 24.5.3.3.14	numarray 26.6
move.iter.op.+ 24.5.3.3.8	numeric.iota 26.7.6
move.iter.op. $+=$ 24.5.3.3.9	numeric.limits 18.3.2.3
move.iter.op 24.5.3.3.10	numeric.limits.members 18.3.2.4
move.iter.op= 24.5.3.3.11	numeric.ops 26.7
move.iter.op.comp 24.5.3.3.13	numeric.ops.overview 26.7.1
move.iter.op.const 24.5.3.3.1	numeric.requirements 26.2
move.iter.op.conv 24.5.3.3.3	numeric.special 18.3.2.7
move.iter.op.decr 24.5.3.3.7	numerics 26
move.iter.op.incr 24.5.3.3.6	numerics.general 26.1
move.iter.op.index 24.5.3.3.12	numeries.generar 20.1
move.iter.op.ref 24.5.3.3.5	O
move.iter.op.rer 24.5.3.3.4	objects.within.classes 17.5.2.3
move.iter.op.scar 24.5.3.3.4 move.iter.op= 24.5.3.3.2	ofstream 27.9.1.10
move.iter.op= 24.5.3.2 move.iter.ops 24.5.3.3	ofstream.assign 27.9.1.12
	ofstream.cons 27.9.1.11
move.iter.requirements 24.5.3.2 move.iterator 24.5.3.1	ofstream.members 27.9.1.13
move.iterators 24.5.3	operators 20.2.1
multibyte.strings 17.5.2.1.4.2	organization 17.6.1
multimap 23.4.5	ostream 27.7.3.1

ostream.assign 27.7.3.3	over.match.oper 13.3.1.2
ostream.cons 27.7.3.2	over.match.ref 13.3.1.6
ostream.formatted 27.7.3.6	over.match.viable 13.3.2
ostream.formatted.reqmts 27.7.3.6.1	over.oper 13.5
ostream.inserters 27.7.3.6.3	over.over 13.4
ostream.inserters.arithmetic 27.7.3.6.2	over.ref $13.5.6$
ostream.inserters.character 27.7.3.6.4	over.sub $13.5.5$
ostream.iterator 24.6.2	over.unary 13.5.1
ostream.iterator.cons.des 24.6.2.1	overflow.error 19.2.8
ostream.iterator.ops 24.6.2.2	
ostream.manip 27.7.3.8	P
ostream.rvalue 27.7.3.9	pair.astuple 20.3.4
ostream.seeks 27.7.3.5	pair.piecewise 20.3.5
ostream.unformatted 27.7.3.7	pairs 20.3
ostream::sentry 27.7.3.4	pairs.general 20.3.1
ostreambuf.iter.cons 24.6.4.1	pairs.pair 20.3.2
ostreambuf.iter.ops 24.6.4.2	pairs.spec 20.3.3
ostreambuf.iterator 24.6.4	partial.sort 25.4.1.3
ostringstream 27.8.4	partial.sort.copy 25.4.1.4
ostringstream.assign 27.8.4.2	partial.sum 26.7.4
ostringstream.cons 27.8.4.1	pointer.traits 20.6.3
ostringstream.members 27.8.4.3	pointer.traits.functions 20.6.3.2
out.of.range 19.2.5	pointer.traits.types 20.6.3.1
output.iterators 24.2.4	pop.heap 25.4.6.2
output.streams 27.7.3	predef.iterators 24.5
over 13	priority.queue 23.6.4
over.ass 13.5.3	priqueue.cons 23.6.4.1
over.best.ics 13.3.3.1	priqueue.cons.alloc 23.6.4.2
over.binary 13.5.2	priqueue.members 23.6.4.3
over.built 13.6	priqueue.special 23.6.4.4
over.call 13.5.4	propagation 18.8.5
over.call.func 13.3.1.1.1	protection.within.classes 17.6.5.10
over.call.object 13.3.1.1.2	ptr.align 20.6.5
over.dcl 13.2	push.heap 25.4.6.1
over.ics.ellipsis 13.3.3.1.3	pasimoup 2001001
over.ics.list 13.3.3.1.5	Q
over.ics.rank 13.3.3.2	queue 23.6.3
over.ics.ref 13.3.3.1.4	queue.cons 23.6.3.2
over.ics.scs 13.3.3.1.1	queue.cons.alloc 23.6.3.3
over.ics.user 13.3.3.1.2	queue.defn 23.6.3.1
over.inc 13.5.7	queue.ops 23.6.3.4
over.literal 13.5.8	queue.special 23.6.3.5
over.load 13.1	queue.syn 23.6.2
over.match 13.3	queue.syn 20.0.2
over.match.best 13.3.3	R
over.match.call 13.3.1.1	$\frac{1}{26.5}$
over.match.conv 13.3.1.5	rand.adapt 26.5.4
over.match.copy 13.3.1.4	rand.adapt.disc 26.5.4.2
over.match.ctor 13.3.1.3	rand.adapt.disc 20.5.4.2 rand.adapt.general 26.5.4.1
over.match.funcs 13.3.1	rand.adapt.ibits 26.5.4.3
over.match.list 13.3.1.7	rand.adapt.shuf 26.5.4.4
Over matchillst 15.5.1.7	ranu.auapt.snur 20.0.4.4

rand.device 26.5.6	ratio.si 20.10.6
rand.dist 26.5.8	ratio.syn 20.10.2
rand.dist.bern 26.5.8.3	re 28
rand.dist.bern.bernoulli 26.5.8.3.1	re.alg 28.11
rand.dist.bern.bin 26.5.8.3.2	re.alg.match 28.11.2
rand.dist.bern.geo 26.5.8.3.3	re.alg.replace 28.11.4
rand.dist.bern.negbin 26.5.8.3.4	re.alg.search 28.11.3
rand.dist.general 26.5.8.1	re.badexp 28.6
rand.dist.norm 26.5.8.5	re.const 28.5
rand.dist.norm.cauchy 26.5.8.5.4	re.def 28.2
rand.dist.norm.chisq 26.5.8.5.3	re.err 28.5.3
rand.dist.norm.f 26.5.8.5.5	re.except 28.11.1
rand.dist.norm.lognormal 26.5.8.5.2	re.general 28.1
rand.dist.norm.normal 26.5.8.5.1	re.grammar 28.13
rand.dist.norm.t 26.5.8.5.6	re.iter 28.12
rand.dist.pois 26.5.8.4	re.matchflag 28.5.2
rand.dist.pois.exp 26.5.8.4.2	re.regex 28.8
rand.dist.pois.extreme 26.5.8.4.5	re.regex.assign 28.8.3
rand.dist.pois.gamma 26.5.8.4.3	re.regex.const 28.8.1
rand.dist.pois.poisson 26.5.8.4.1	re.regex.construct 28.8.2
rand.dist.pois.weibull 26.5.8.4.4	re.regex.locale 28.8.5
rand.dist.samp 26.5.8.6	re.regex.nmswap 28.8.7.1
rand.dist.samp.discrete 26.5.8.6.1	re.regex.nonmemb 28.8.7
rand.dist.samp.pconst 26.5.8.6.2	re.regex.operations 28.8.4
rand.dist.samp.plinear 26.5.8.6.3	re.regex.swap 28.8.6
rand.dist.uni 26.5.8.2	re.regiter 28.12.1
rand.dist.uni.int 26.5.8.2.1	re.regiter.cnstr 28.12.1.1
rand.dist.uni.real 26.5.8.2.2	re.regiter.comp 28.12.1.2
rand.eng 26.5.3	re.regiter.deref 28.12.1.3
rand.eng.lcong 26.5.3.1	re.regiter.incr 28.12.1.4
rand.eng.mers 26.5.3.2	re.req 28.3
rand.eng.sub 26.5.3.3	re.results 28.10
rand.predef 26.5.5	re.results.acc 28.10.4
rand.req 26.5.1	re.results.all 28.10.6
rand.req.adapt 26.5.1.5	re.results.const 28.10.1
rand.req.dist 26.5.1.6	re.results.form 28.10.5
rand.req.eng 26.5.1.4	re.results.nonmember 28.10.8
rand.req.genl 26.5.1.1	re.results.size 28.10.3
rand.req.seedseq 26.5.1.2	re.results.state 28.10.2
rand.req.urng 26.5.1.3	re.results.swap 28.10.7
rand.synopsis 26.5.2	re.submatch 28.9
rand.util 26.5.7	re.submatch.members 28.9.1
rand.util.canonical 26.5.7.2	re.submatch.op 28.9.2
rand.util.seedseq 26.5.7.1	re.syn 28.4
random.access.iterators 24.2.7	re.synopt 28.5.1
range.error 19.2.7	re.tokiter 28.12.2
ratio 20.10	
ratio 20.10 ratio.arithmetic 20.10.4	re.tokiter.cnstr 28.12.2.1
	re.tokiter.comp 28.12.2.2
ratio.comparison 20.10.5	re.tokiter.deref 28.12.2.3
ratio general 20.10.1	re.tokiter.incr 28.12.2.4
ratio.ratio 20.10.3	re.traits 28.7

reentrancy 17.6.5.8	set 23.4.6
refwrap 20.8.3	set.cons 23.4.6.2
refwrap.access 20.8.3.3	set.difference 25.4.5.4
refwrap.assign 20.8.3.2	set.intersection 25.4.5.3
refwrap.const 20.8.3.1	set.new.handler 18.6.2.4
refwrap.helpers 20.8.3.5	set.overview 23.4.6.1
refwrap.invoke 20.8.3.4	set.special 23.4.6.3
replacement.functions 17.6.4.6	set.symmetric.difference 25.4.5.5
requirements 17.6	set.terminate 18.8.3.2
res.on.arguments 17.6.4.9	set.unexpected D.11.2
res.on.data.races 17.6.5.9	set.union 25.4.5.2
res.on.exception.handling 17.6.5.12	slice.access 26.6.4.3
res.on.functions 17.6.4.8	slice.arr.assign 26.6.5.2
res.on.headers 17.6.5.2	slice.arr.comp.assign 26.6.5.3
res.on.macro.definitions 17.6.5.3	slice.arr.fill $26.6.5.4$
res.on.objects 17.6.4.10	smartptr 20.7
res.on.pointer.storage 17.6.5.13	sort 25.4.1.1
res.on.required 17.6.4.11	sort.heap 25.4.6.4
reserved.names 17.6.4.3	special 12
reverse.iter.cons 24.5.1.3.1	specialized.addressof 20.6.12.1
reverse.iter.conv 24.5.1.3.3	specialized.algorithms 20.6.12
reverse.iter.op!= $24.5.1.3.15$	stable.sort 25.4.1.2
reverse.iter.op+ 24.5.1.3.8	stack 23.6.5
reverse.iter.op++ $24.5.1.3.6$	stack.cons 23.6.5.3
reverse.iter.op+= $24.5.1.3.9$	stack.cons.alloc 23.6.5.4
reverse.iter.op- 24.5.1.3.10	stack.defn 23.6.5.2
reverse.iter.op= $24.5.1.3.11$	stack.ops 23.6.5.5
reverse.iter.op.star 24.5.1.3.4	stack.special 23.6.5.6
reverse.iter.op $<$ 24.5.1.3.14	stack.syn 23.6.5.1
reverse.iter.op $<=$ 24.5.1.3.18	std.exceptions 19.2
reverse.iter.op= $24.5.1.3.2$	std.ios.manip 27.5.6
reverse.iter.op== $24.5.1.3.13$	std.iterator.tags 24.4.3
reverse.iter.op> 24.5.1.3.16	std.manip 27.7.4
reverse.iter.op>= $24.5.1.3.17$	stmt.ambig 6.8
reverse.iter.opdiff 24.5.1.3.19	stmt.block 6.3
reverse.iter.opindex 24.5.1.3.12	stmt.break 6.6.1
reverse.iter.opref 24.5.1.3.5	stmt.cont 6.6.2
reverse.iter.ops 24.5.1.3	stmt.dcl 6.7
reverse.iter.opsum 24.5.1.3.20	stmt.do 6.5.2
reverse.iter.op 24.5.1.3.7	stmt.expr 6.2
reverse.iter.requirements 24.5.1.2	stmt.for 6.5.3
reverse.iterator 24.5.1.1	stmt.goto 6.6.4
reverse.iterators 24.5.1	stmt.if 6.4.1
round.style 18.3.2.5	stmt.iter 6.5
runtime.error 19.2.6	stmt.jump 6.6
	stmt.label 6.1
$\mathbf{S}$	stmt.ranged 6.5.4
scoped.adaptor.operators 20.12.5	stmt.return 6.6.3
sequence.reqmts 23.2.3	stmt.select 6.4
sequences 23.3	stmt.stmt 6
sequences.general 23.3.1	stmt.switch 6.4.2
	-

stmt.while 6.5.1	string::op!= $21.4.8.3$
storage.iterator 20.6.10	string::op+ 21.4.8.1
stream.buffers 27.6	string::op+= 21.4.6.1
stream.buffers.overview 27.6.1	string::op< $21.4.8.4$
stream.iterators 24.6	string::op <= 21.4.8.6
stream.types 27.5.2	string::op> $21.4.8.5$
streambuf 27.6.3	string::op>= 21.4.8.7
streambuf.assign 27.6.3.3.1	string::operator == 21.4.8.2
streambuf.buffer 27.6.3.2.2	string::replace 21.4.6.6
streambuf.cons 27.6.3.1	string::rfind 21.4.7.3
streambuf.get.area 27.6.3.3.2	string::substr 21.4.7.8
streambuf.locales 27.6.3.2.1	string::swap 21.4.6.8
streambuf.members 27.6.3.2	stringbuf 27.8.2
streambuf.protected 27.6.3.3	stringbuf.assign 27.8.2.2
streambuf.pub.get 27.6.3.2.3	stringbuf.cons 27.8.2.1
streambuf.pub.pback 27.6.3.2.4	stringbuf.members 27.8.2.3
streambuf.pub.put 27.6.3.2.5	stringbuf.virtuals 27.8.2.4
streambuf.put.area 27.6.3.3.3	strings 21
streambuf.regts 27.6.2	strings.general 21.1
streambuf.virt.buffer 27.6.3.4.2	stringstream 27.8.5
streambuf.virt.get 27.6.3.4.3	stringstream.assign 27.8.6.1
streambuf.virt.locales 27.6.3.4.1	stringstream.cons 27.8.6
streambuf.virt.pback 27.6.3.4.4	stringstream.members 27.8.7
streambuf.virt.put 27.6.3.4.5	structure 17.5.1
streambuf.virtuals 27.6.3.4	structure.elements 17.5.1.1
string.access 21.4.5	structure.requirements 17.5.1.3
string.accessors 21.4.7.1	structure.see.also 17.5.1.5
string.capacity 21.4.4	structure.specifications 17.5.1.4
string.classes 21.3	structure.summary 17.5.1.2
string.cons 21.4.2	support.dynamic 18.6
string.conversions 21.5	support.exception 18.8
string.io 21.4.8.9	support.general 18.1
string.iterators 21.4.3	support.initlist 18.9
string.modifiers 21.4.6	support.initiist 18.3 support.initiist.access 18.9.2
string.nonmembers 21.4.8	support.initist.access 18.9.1
string.ops 21.4.7	support.initist.cons 18.9.1 support.initlist.range 18.9.3
string.require 21.4.1	support.limits 18.3
string.require 21.4.1 string.special 21.4.8.8	support.limits 18.3 support.limits.general 18.3.1
string.streams 27.8	support.rtti 18.7
~	
string.streams.overview 27.8.1	support.runtime 18.10
string::append 21.4.6.2	support.start.term 18.5
string::assign 21.4.6.3	support.types 18.2
string::compare 21.4.7.9	swappable.requirements 17.6.3.2
string::copy 21.4.6.7	syntax 1.6
string::erase 21.4.6.5	syserr 19.5
string::find 21.4.7.2	syserr.compare 19.5.4
string::find.first.not.of 21.4.7.6	syserr.errcat 19.5.1
string::find.first.of 21.4.7.4	syserr.errcat.derived 19.5.1.4
string::find.last.not.of 21.4.7.7	syserr.errcat.nonvirtuals 19.5.1.3
	the contract of the contract o
string::find.last.of 21.4.7.5	syserr.errcat.objects 19.5.1.5
string::insert 21.4.6.4	syserr.errcat.objects 19.5.1.5 syserr.errcat.overview 19.5.1.1

syserr.errcat.virtuals 19.5.1.2	temp.func.order 14.5.6.2
syserr.errcode 19.5.2	temp.inject 14.6.5
syserr.errcode.constructors 19.5.2.2	temp.inst 14.7.1
syserr.errcode.modifiers 19.5.2.3	temp.local 14.6.1
syserr.errcode.nonmembers 19.5.2.5	temp.mem $14.5.2$
syserr.errcode.observers 19.5.2.4	temp.mem.class 14.5.1.2
syserr.errcode.overview 19.5.2.1	temp.mem.enum 14.5.1.4
syserr.errcondition 19.5.3	temp.mem.func 14.5.1.1
syserr.errcondition.constructors 19.5.3.2	temp.names 14.2
syserr.errcondition.modifiers 19.5.3.3	temp.nondep 14.6.3
syserr.errcondition.nonmembers 19.5.3.5	temp.over 14.8.3
syserr.errcondition.observers 19.5.3.4	temp.over.link $14.5.6.1$
syserr.errcondition.overview 19.5.3.1	temp.param 14.1
syserr.hash 19.5.5	temp.point $14.6.4.1$
syserr.syserr 19.5.6	temp.res $14.6$
syserr.syserr.members 19.5.6.2	temp.spec 14.7
syserr.syserr.overview 19.5.6.1	temp.static 14.5.1.3
	temp.type 14.4
${f T}$	temp.variadic 14.5.3
temp = 14	template.bitset 20.5
temp.alias 14.5.7	template.gslice.array 26.6.7
temp.arg 14.3	template.gslice.array.overview 26.6.7.1
temp.arg.explicit 14.8.1	template.indirect.array 26.6.9
temp.arg.nontype 14.3.2	template.indirect.array.overview 26.6.9.1
temp.arg.template 14.3.3	template.mask.array 26.6.8
temp.arg.type 14.3.1	template.mask.array.overview 26.6.8.1
temp.class 14.5.1	template.slice.array 26.6.5
temp.class.order 14.5.5.2	template.slice.array.overview 26.6.5.1
temp.class.spec 14.5.5	template.valarray 26.6.2
temp.class.spec.match 14.5.5.1	template.valarray.overview 26.6.2.1
temp.class.spec.mfunc 14.5.5.3	temporary.buffer 20.6.11
temp.decls 14.5	terminate 18.8.3.4
temp.deduct 14.8.2	terminate.handler 18.8.3.1
temp.deduct.call 14.8.2.1	thread 30
temp.deduct.conv 14.8.2.3	thread.condition 30.5
temp.deduct.decl 14.8.2.6	thread.condition.condvar 30.5.1
temp.deduct.funcaddr 14.8.2.2	thread.condition.condvarany 30.5.2
temp.deduct.partial 14.8.2.4	thread.decaycopy 30.2.6
temp.deduct.type 14.8.2.5	thread.general 30.1
temp.dep 14.6.2	thread.lock 30.4.2
temp.dep.candidate 14.6.4.2	thread.lock.algorithm 30.4.3
temp.dep.constexpr 14.6.2.3	thread.lock.guard 30.4.2.1
temp.dep.expr 14.6.2.2	thread.lock.unique 30.4.2.2
temp.dep.res 14.6.4	thread.lock.unique.cons 30.4.2.2.1
temp.dep.temp 14.6.2.4	thread.lock.unique.locking 30.4.2.2.2
temp.dep.type 14.6.2.1	thread.lock.unique.mod 30.4.2.2.3
temp.expl.spec 14.7.3	thread.lock.unique.obs 30.4.2.2.4
temp.explicit 14.7.2	thread.mutex 30.4
temp.fct 14.5.6	thread.mutex.class 30.4.1.2.1
temp.fct.spec 14.8	thread.mutex.recursive 30.4.1.2.2
temp.friend 14.5.4	thread.mutex.requirements 30.4.1
	om sadmidom oqui om om o

thread.mutex.requirements.general 30.4.1.1	time.syn $20.11.2$
thread.mutex.requirements.mutex 30.4.1.2	time.traits 20.11.4
thread.once 30.4.4	time.traits.duration_values 20.11.4.2
thread.once.callonce 30.4.4.2	time.traits.is_fp 20.11.4.1
thread.once.onceflag 30.4.4.1	time.traits.specializations 20.11.4.3
thread.req 30.2	tuple 20.4
thread.req.exception 30.2.2	tuple.assign 20.4.2.2
thread.req.lockable 30.2.5	tuple.cnstr 20.4.2.1
thread.req.lockable.basic 30.2.5.2	tuple.creation 20.4.2.4
thread.req.lockable.general 30.2.5.1	tuple.elem 20.4.2.6
thread.req.lockable.req 30.2.5.3	tuple.general 20.4.1
thread.req.lockable.timed 30.2.5.4	tuple.helper 20.4.2.5
thread.req.native 30.2.3	tuple.rel 20.4.2.7
thread.req.paramname 30.2.1	tuple.special 20.4.2.9
thread.req.timing 30.2.4	tuple.swap 20.4.2.3
thread.thread.algorithm 30.3.1.7	tuple.traits 20.4.2.8
thread.thread.assign 30.3.1.4	tuple.tuple 20.4.2
thread.thread.class 30.3.1	type.descriptions 17.5.2.1
thread.thread.constr 30.3.1.2	type.descriptions.general 17.5.2.1.1
thread.thread.destr 30.3.1.3	type.index 20.13
thread.thread.id 30.3.1.1	type.index.hash 20.13.4
thread.thread.member 30.3.1.5	type.index.members 20.13.3
thread.thread.static 30.3.1.6	type.index.overview 20.13.2
thread, thread, this 30.3.2	type.index.synopsis 20.13.1
thread.threads 30.3	type.info 18.7.1
thread.timedmutex.class 30.4.1.3.1	type.iiiio 10.11.1
thread.timedmutex.recursive 30.4.1.3.2	$\mathbf{U}$
thread.timedmutex.recursive 30.4.1.3.2 thread.timedmutex.requirements 30.4.1.3	U uncaught 18.8.4
thread.timedmutex.requirements 30.4.1.3	uncaught 18.8.4
thread.timed mutex.requirements $30.4.1.3$ time $20.11$	uncaught 18.8.4 underflow.error 19.2.9
thread.timed mutex.requirements $30.4.1.3$ time $20.11$ time.clock $20.11.7$	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4
thread.timed mutex.requirements $30.4.1.3$ time $20.11$ time.clock $20.11.7$ time.clock.hires $20.11.7.3$	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1
thread.timed mutex.requirements $30.4.1.3$ time $20.11$ time.clock $20.11.7$ time.clock.hires $20.11.7.3$ time.clock.req $20.11.3$	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2
thread.timedmutex.requirements $30.4.1.3$ time $20.11$ time.clock $20.11.7$ time.clock.hires $20.11.7.3$ time.clock.req $20.11.3$ time.clock.steady $20.11.7.2$	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3
thread.timedmutex.requirements $30.4.1.3$ time $20.11$ time.clock $20.11.7$ time.clock.hires $20.11.7.3$ time.clock.req $20.11.3$ time.clock.steady $20.11.7.2$ time.clock.system $20.11.7.1$	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4
thread.timedmutex.requirements $30.4.1.3$ time $20.11$ time.clock $20.11.7$ time.clock.hires $20.11.7.3$ time.clock.req $20.11.3$ time.clock.steady $20.11.7.2$ time.clock.system $20.11.7.1$ time.duration $20.11.5$	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2
thread.timedmutex.requirements $30.4.1.3$ time $20.11$ time.clock $20.11.7$ time.clock.hires $20.11.7.3$ time.clock.req $20.11.3$ time.clock.steady $20.11.7.2$ time.clock.steady $20.11.7.2$ time.clock.system $20.11.7.1$ time.duration $20.11.5$ time.duration.arithmetic $20.11.5.3$ time.duration.cast $20.11.5.7$ time.duration.comparisons $20.11.5.6$	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.cons 20.11.5.1	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.cons 20.11.5.1 time.duration.nonmember 20.11.5.5	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.cons 20.11.5.1 time.duration.nonmember 20.11.5.5 time.duration.observer 20.11.5.2	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime.ctor 20.7.1.3.1
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.comparisons 20.11.5.1 time.duration.nonmember 20.11.5.5 time.duration.observer 20.11.5.2 time.duration.special 20.11.5.4	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime.ctor 20.7.1.3.1 unique.ptr.runtime.modifiers 20.7.1.3.3
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.comparisons 20.11.5.1 time.duration.nonmember 20.11.5.5 time.duration.observer 20.11.5.2 time.duration.special 20.11.5.4 time.general 20.11.1	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime.ctor 20.7.1.3.1 unique.ptr.runtime.modifiers 20.7.1.3.3 unique.ptr.runtime.modifiers 20.7.1.3.3 unique.ptr.runtime.observers 20.7.1.3.2
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.comparisons 20.11.5.1 time.duration.nonmember 20.11.5.5 time.duration.observer 20.11.5.2 time.duration.special 20.11.5.4 time.general 20.11.1	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime.ctor 20.7.1.3.1 unique.ptr.runtime.modifiers 20.7.1.3.3 unique.ptr.runtime.observers 20.7.1.3.2 unique.ptr.single 20.7.1.2
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cost 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.nonmember 20.11.5.5 time.duration.observer 20.11.5.2 time.duration.special 20.11.5.4 time.general 20.11.1 time.point 20.11.6 time.point.arithmetic 20.11.6.3	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime.ctor 20.7.1.3.1 unique.ptr.runtime.modifiers 20.7.1.3.3 unique.ptr.runtime.observers 20.7.1.3.2 unique.ptr.single 20.7.1.2 unique.ptr.single.asgn 20.7.1.2.3
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.nonmember 20.11.5.5 time.duration.observer 20.11.5.2 time.duration.observer 20.11.5.2 time.duration.special 20.11.5.4 time.general 20.11.6 time.point 20.11.6 time.point.arithmetic 20.11.6.3 time.point.cast 20.11.6.7	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime.ctor 20.7.1.3.1 unique.ptr.runtime.modifiers 20.7.1.3.2 unique.ptr.runtime.observers 20.7.1.3.2 unique.ptr.single 20.7.1.2 unique.ptr.single.asgn 20.7.1.2.3 unique.ptr.single.asgn 20.7.1.2.3 unique.ptr.single.ctor 20.7.1.2.1
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.comparisons 20.11.5.1 time.duration.nonmember 20.11.5.5 time.duration.observer 20.11.5.2 time.duration.special 20.11.5.4 time.general 20.11.1 time.point 20.11.6 time.point.cast 20.11.6.7 time.point.comparisons 20.11.6.6	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime.ctor 20.7.1.3.1 unique.ptr.runtime.modifiers 20.7.1.3.3 unique.ptr.runtime.observers 20.7.1.3.2 unique.ptr.single 20.7.1.2 unique.ptr.single.asgn 20.7.1.2.3 unique.ptr.single.ctor 20.7.1.2.1 unique.ptr.single.dtor 20.7.1.2.2
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.comparisons 20.11.5.5 time.duration.nonmember 20.11.5.5 time.duration.observer 20.11.5.2 time.duration.special 20.11.5.4 time.general 20.11.1 time.point 20.11.6 time.point.arithmetic 20.11.6.3 time.point.comparisons 20.11.6.6 time.point.comparisons 20.11.6.1	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime.ctor 20.7.1.3.1 unique.ptr.runtime.modifiers 20.7.1.3.3 unique.ptr.runtime.observers 20.7.1.3.2 unique.ptr.single 20.7.1.2 unique.ptr.single.asgn 20.7.1.2.3 unique.ptr.single.ctor 20.7.1.2.1 unique.ptr.single.dtor 20.7.1.2.2 unique.ptr.single.modifiers 20.7.1.2.5
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.nonmember 20.11.5.5 time.duration.observer 20.11.5.2 time.duration.observer 20.11.5.4 time.general 20.11.1 time.point 20.11.6 time.point.arithmetic 20.11.6.3 time.point.comparisons 20.11.6.6 time.point.comparisons 20.11.6.1 time.point.nonmember 20.11.6.5	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime.ctor 20.7.1.3.1 unique.ptr.runtime.modifiers 20.7.1.3.3 unique.ptr.runtime.observers 20.7.1.3.2 unique.ptr.single 20.7.1.2 unique.ptr.single.asgn 20.7.1.2.3 unique.ptr.single.ctor 20.7.1.2.1 unique.ptr.single.dtor 20.7.1.2.2 unique.ptr.single.modifiers 20.7.1.2.5 unique.ptr.single.observers 20.7.1.2.4
thread.timedmutex.requirements 30.4.1.3 time 20.11 time.clock 20.11.7 time.clock.hires 20.11.7.3 time.clock.req 20.11.3 time.clock.steady 20.11.7.2 time.clock.system 20.11.7.1 time.duration 20.11.5 time.duration.arithmetic 20.11.5.3 time.duration.cast 20.11.5.7 time.duration.comparisons 20.11.5.6 time.duration.comparisons 20.11.5.5 time.duration.nonmember 20.11.5.5 time.duration.observer 20.11.5.2 time.duration.special 20.11.5.4 time.general 20.11.1 time.point 20.11.6 time.point.arithmetic 20.11.6.3 time.point.comparisons 20.11.6.6 time.point.comparisons 20.11.6.1	uncaught 18.8.4 underflow.error 19.2.9 unexpected D.11.4 unexpected.handler D.11.1 uninitialized.copy 20.6.12.2 uninitialized.fill 20.6.12.3 uninitialized.fill.n 20.6.12.4 unique.ptr 20.7.1 unique.ptr.dltr 20.7.1.1 unique.ptr.dltr.dflt 20.7.1.1.2 unique.ptr.dltr.dflt1 20.7.1.1.3 unique.ptr.dltr.general 20.7.1.1.1 unique.ptr.runtime 20.7.1.3 unique.ptr.runtime.ctor 20.7.1.3.1 unique.ptr.runtime.modifiers 20.7.1.3.3 unique.ptr.runtime.observers 20.7.1.3.2 unique.ptr.single 20.7.1.2 unique.ptr.single.asgn 20.7.1.2.3 unique.ptr.single.ctor 20.7.1.2.1 unique.ptr.single.dtor 20.7.1.2.2 unique.ptr.single.modifiers 20.7.1.2.5

unord.general 23.5.1	util.smartptr.weak 20.7.2.3
unord.hash 20.8.12	util.smartptr.weak.assign 20.7.2.3.3
unord.map 23.5.4	util.smartptr.weak.const 20.7.2.3.1
unord.map.cnstr 23.5.4.2	util.smartptr.weak.dest 20.7.2.3.2
unord.map.elem 23.5.4.3	util.smartptr.weak.mod 20.7.2.3.4
unord.map.modifiers 23.5.4.4	util.smartptr.weak.obs 20.7.2.3.5
unord.map.overview 23.5.4.1	util.smartptr.weak.spec 20.7.2.3.6
unord.map.swap 23.5.4.5	util.smartptr.weakptr 20.7.2.1
unord.map.syn 23.5.2	utilities 20
unord.multimap 23.5.5	utilities.general 20.1
unord.multimap.cnstr 23.5.5.2	utility 20.2
unord.multimap.modifiers 23.5.5.3	utility.arg.requirements 17.6.3.1
unord.multimap.overview 23.5.5.1	utility.requirements 17.6.3
unord.multimap.swap 23.5.5.4	utility.swap 20.2.2
unord.multiset 23.5.7	
unord.multiset.cnstr 23.5.7.2	$\mathbf{V}$
unord.multiset.overview 23.5.7.1	valarray.access 26.6.2.4
unord.multiset.swap 23.5.7.3	valarray.assign 26.6.2.3
unord.req 23.2.5	valarray.binary 26.6.3.1
unord.req.except 23.2.5.1	valarray.cassign 26.6.2.7
unord.set 23.5.6	valarray.comparison 26.6.3.2
unord.set.cnstr 23.5.6.2	valarray.cons 26.6.2.2
unord.set.overview 23.5.6.1	valarray.members 26.6.2.8
unord.set.swap 23.5.6.3	valarray.nonmembers 26.6.3
unord.set.syn 23.5.3	valarray.range 26.6.10
upper.bound 25.4.3.2	valarray.range 26.6.3.4
using 17.6.2	valarray.sub 26.6.2.5
using.headers 17.6.2.2	valarray.syn 26.6.1
using.linkage 17.6.2.3	valarray.transcend 26.6.3.3
using.overview 17.6.2.1	valarray.unary 26.6.2.6
usrlit.suffix 17.6.4.3.5	value.error.codes 17.6.5.14
util.dynamic.safety 20.6.4	vector 23.3.6
util.smartptr 20.7.2	vector 25.5.6 vector.bool 23.3.7
util.smartptr.enab 20.7.2.4	vector.capacity 23.3.6.3
util.smartptr.getdeleter 20.7.2.2.10	vector.cons 23.3.6.2
util.smartptr.hash 20.7.2.6	vector.data 23.3.6.4
util.smartptr.ownerless 20.7.2.3.7	vector.data 25.5.0.4 vector.modifiers 23.3.6.5
util.smartptr.shared 20.7.2.2	
util.smartptr.shared.assign 20.7.2.2.3	vector.overview 23.3.6.1 vector.special 23.3.6.6
util.smartptr.shared.atomic 20.7.2.5	vector.special 23.3.6.6
util.smartptr.shared.cast 20.7.2.2.9	***
util.smartptr.shared.cmp 20.7.2.2.7	$\mathbf{W}$
util.smartptr.shared.const 20.7.2.2.1	wide.stream.objects 27.4.3
util.smartptr.shared.create 20.7.2.2.6	
util.smartptr.shared.dest 20.7.2.2.2	$\mathbf{X}$
util.smartptr.shared.io 20.7.2.2.11	$\mathbf{xref}  \mathbf{F}$
util.smartptr.shared.mod 20.7.2.2.4	
util.smartptr.shared.obs 20.7.2.2.5	Y
util.smartptr.shared.spec 20.7.2.2.8	${f z}$
adilities open and a second se	_

# Index

!, see operator, logical negation	#if, $388$ , $422$
!=, see inequality operator	#11, 389, 422 #ifdef, 389
(), see operator, function call, see declarator, func-	#ifndef, 389
tion	#include, 389, 409
*, see operator, indirection, see multiplication op-	#line, see preprocessing directives, line control
erator, see declarator, pointer	#pragma, see preprocessing directives, ragma
+, see operator, unary plus, see addition operator	#undef, 393, 419
++, see operator, increment	%, see remainder operator
	&, see operator, address-of, see bitwise AND open
,, see comma operator -, see operator, unary minus, see subtraction op-	ator, see declarator, reference
erator	&&, see logical AND operator
->, see operator, class member access	^, see bitwise exclusive OR operator
->*, see operator, class member access	DATE, 396
, see operator, decrement	FILE, 396
., see operator, class member access	LINE, 396
.*, see pointer to member operator	STDC, 396
, see pointer to member operator, see ellipsis	implementation-defined, 396
/, see division operator	STDCPP_STRICT_POINTER_SAFETY, 397
, see division operator	implementation-defined, 397
field declaration, 216	STDCPP_THREADS, 397
label specifier, 124	implementation-defined, 397
::, see scope resolution operator	STDC_HOSTED, 396
::*, see declarator, pointer to member	implementation-defined, 396
<, see less than operator	STDC_ISO_10646, 397
template and, 306, 307	implementation-defined, 397
<>, see preprocessing directive, header	STDC_MB_MIGHT_NEQ_WC, 396
<=, see less than or equal to operator	implementation-defined, 396
<, see left shift operator	STDC_VERSION, 396
=, see assignment operator	implementation-defined, 396
==, see equality operator	TIME, 396
>, see greater than operator	VA_ARGS, 391
>=, see greater than or equal operator	cplusplus, 396
>>, see greater than or equal operator	see backslash
?:, see conditional expression operator	{}
[], see operator, subscripting, see declarator, ar-	block statement, 124
ray	class declaration, 204
"", see preprocessing directives, source-file in-	class definition, 204
clusion	enum declaration, 149
# operator, 391, 392	initializer list, 193
## operator, 392	~, see destructor
#define, 391	_, see character, underscore
#elif, 388	, see bitwise inclusive OR operator
#else, 389	, see logical OR operator
#endif, 389	~, see operator, one's complement
#error, see preprocessing directives, error	, and a parametr, one of comprehensive
, 300 proprocessing anounting, error	0, see also zero, null

null character, 29	unspecified, 209
string terminator, 29	allocation functions, 62
	allowing an exception, see exception handling, al-
abort, 62, 129	lowing an exception
abstract-declarator, 173, 1175	alternative token, see token, alternative
abstract-pack-declarator, 173, 1175	ambiguity
access control, 231–241	base class member, 222
base class, 233	class conversion, 224
base class member, 219	declaration type, 135
class member, 96	declaration versus cast, 174
default, 231	declaration versus expression, 131
default argument, 232	function declaration, 191
friend function, 236	member access, 222
member name, 231	overloaded function, 276
multiple access, 240	parentheses and, 107
nested class, 240	Amendment 1, 419
overloading and, 276	and-expression, 116, 1168
private, 231	appertain, 168
protected, 231, 239	argc, 58
public, 231	argument, 2, 421, 422, 458
union default member, 204	access checking and default, 232
using-declaration and, 161	binding of default, 185
virtual function, 240	
access-specifier, 219, 1177	evaluation of default, 185, 186
access control	example of default, 184, 185
anonymous union, 215	function call expression, 2
member function and, 242	function-like macro, 2
overloading resolution and, 223	overloaded operator and default, 296
access specifier, 232, 233	reference, 94
addition operator, 113	scope of default, 186
	template, 308
additive-expression, 113, 1168	template instantiation, 2
address, 73, 116 address of member function	throw expression, 2
	type checking of default, 185
unspecified, 422	arguments
aggregate, 193	implementation-defined order of evaluation of
aggregate initialization, 193	function, 186
algorithm	argument and name hiding
stable, 401, 423	default, 186
alias	argument and virtual function
namespace, 156	default, 187
alias template, 331	argument list
alias-declaration, 133, 1170	empty, 181
alignment	variable, 181
extended, 76	argument passing, 94
fundamental, 75	reference and, 197
alignment-specifier, 167, 1173	argument substitution, see macro, argument sub-
alignment requirement	stitution
implementation-defined, 75	argument type
allocation	unknown, 181
alignment storage, 108	argv, 58
implementation defined bit-field, 216	arithmetic

pointer, 113	exception safety, 718
unsigned, 71	requirements, 718
array, 182	unordered, see unordered associative contain-
bound, 179	ers
const, 73	asynchronous provider, 1146
delete, 110	asynchronous return object, 1146
multidimensional, 180	atexit, 61
new, 107	atomic operations, see operation, atomic
overloading and pointer versus, 274	attribute, 167–171
sizeof, 105	alignment, 168
storage of, 181	carries dependency, 170
array	noreturn, 169
as aggregate, 730	syntax and semantics, 167
contiguous storage, 730	attribute, 167, 1173
initialization, 730, 731	attribute-argument-clause, 168, 1174
tuple interface to, 732	attribute-declaration, 133, 1170
zero sized, 732	attribute-list, 167, 1173
array size	attribute-namespace, $168$ , $1174$
default, 180	attribute-scoped-token, 168, 1174
	attribute-specifier, 167, 1173
arrow operator, see operator, class member access	attribute-specifier-seq, 167, 1173
as-if rule, 8	- v -: ·
asm	attribute-token, 167, 1174
implementation-defined, 164	automatic storage duration, 62
asm-definition, 164, 1173	$\mathtt{awk},1058$
assembler, 164	backslash character, 26
<assert.h>, 409</assert.h>	bad_alloc, 108
assignment	bad_cast, 98
and Ivalue, 119	
conversion by, 119	bad_exception, 386
copy, see assignment operator, copy	bad_typeid, 99
move, see assignment operator, move, 400	bad_typeid::what
reference, 197	implementation-defined, 447
assignment operator	balanced-token, 168, 1174
copy, 242, 265–267	balanced-token-seq, 168, 1174
hidden, 266	base class
implicitly declared, 265	overloading and, 275
implicitly defined, 267	base class subobject, 7
inaccessible, 267	base-clause, 219, 1176
trivial, 266	base-specifier, 219, 1177
virtual bases and, 267	base-specifier-list, 219, 1177
move, 242, 265–267	base-type-specifier, 219, 1177
hidden, 266	BaseCharacteristic, 559
implicitly declared, 265	base class, 219, 220
implicitly defined, 267	direct, 219
inaccessible, 267	indirect, 219
trivial, 266	private, $233$
virtual bases and, 267	protected, 233
overloaded, 297	public, 233
assignment-expression, 119, 1169	base class virtual, see virtual base class
assignment-operator, 119, 1169	basic_ios::failure argument
associative containers	implementation-defined, 980

begin	braced-init-list, 190, 1176
unordered associative containers, 726	bucket
behavior	unordered associative containers, 726
conditionally-supported, 2, 5	bucket_count
default, 400, 404	unordered associative containers, 725
implementation-defined, 3, 8	bucket_size
locale-specific, 3	unordered associative containers, 726
observable, 8, 9	buckets, 719
on receipt of signal, 8	byte, 6, 105
required, 401, 404	
undefined, $4, 5, 8$	$^{\mathrm{C}}$
unspecified, 4, 8	linkage to, 165
Ben, 276	standard, $1$
Bernoulli distributions, 912–915	standard library, 1
bernoulli_distribution	Unicode TR, 1
discrete probability function, 912	c-char, 25, 1163
binary function, 544	c-char-sequence, 25, 1163
binary operator	call
overloaded, 297	operator function, 296
BinaryTypeTrait, 559	pseudo destructor, 96
binary operator	call signature, 543
interpretation of, 297	call wrapper, 543, 544
bind directly, 199	forwarding, 544
binding	simple, 544
reference, 197	type, 543
binomial_distribution	Callable, 555
discrete probability function, 913	callable object, 543, 555
bit-field, 216	callable type, 543
address of, 216	capture, 88, 1166
alignment of, 216	capture-default, 88, 1166
implementation-defined sign of, 216	capture-list, 88, 1166
implementation defined alignment of, 216	captured, 90
type of, 216	by copy, 91
unnamed, 216	by reference, 91
zero width of, 216	carries a dependency, 12
block, 399	carry
initialization in, 130	subtract_with_carry_engine, 900
block scope, 38	<pre><cassert>, 409</cassert></pre>
block statement, see statement, compound	cast
block-declaration, 133, 1170	base class, 100
block structure, 130	const, 102, 111
body	derived class, 100
function, 187	dynamic, 97, 446
Boolean, 216	construction and, 261
Boolean literal, 29	destruction and, 261
boolean literal, see literal, boolean	integer to pointer, 101
boolean-literal, 29, 1165	lvalue, 99, 101
Boolean type, 71	pointer-to-function, 101
bound arguments, 551	pointer-to-member, 101, 102
bound, of array, 179	pointer to integer, 101
brace-or-equal-initializer, 190, 1175	reference, 99, 102
014cc-01-equal-tititianizet, 190, 1179	1010101100, 00, 102

reinterpret, 101, 111 integer to pointer, 101 lvalue, 101 pointer to integer, 101 pointer-to-function, 101	abstract, 229 base, 419, 424 cast to incomplete, 112 constructor and abstract, 230 definition, 34
pointer-to-member, 102	derived, $424$
reference, 102	linkage of, 56
static, 99, 111	linkage specification, 166
lvalue, 99	member function, see member function, class
reference, 99	pointer to abstract, 230
undefined pointer-to-function, 101	polymorphic, 225
cast-expression, 111, 1168	scope of enumerator, 152
casting, 95	standard-layout, 205
catch, 377	trivial, 204
cauchy_distribution	unnamed, 139
probability density function, 922	class-head, 204, 1176
cbegin	class-head-name, 204, 1176
unordered associative containers, 726	class-key, 204, 1176
cend	class-name, 204, 1176
unordered associative containers, 726	class-or-decltype, 219, 1177
<pre><cerrno>, 419</cerrno></pre>	class-specifier, 204, 1176
char	class-virt-specifier, 204, 1176
implementation-defined sign of, 71	class local, see local class
char-like object, 601	class name, 173
char-like type, 601	elaborated, 148, 206, 207
char16_t, 25	point of declaration, 207
char16_t character, 25	scope of, $206$
char32_t, 25	typedef, $139, 207$
char32_t character, 25	class nested, see nested class
char_class_type	class object
regular expression traits, 1049	assignment to, 119
character, 399	const, $73$
decimal-point, 406	member, 209
multibyte, 3	$\mathtt{sizeof},105$
signed, 71	class object copy, see copy constructor
source file, 16	class object initialization, see constructor
underscore, 419	clear
in identifier, 22	unordered associative containers, 725
character literal, see literal, character	$\langle clocale \rangle$ , $406$
character set, 17–18	closure object, 88
basic execution, 6	closure type, 88
basic source, 16, 17	collating element, 1048
character string literal, 392	comment, 19–20
character-literal, 25, 1163	/* */, 20
character string, 28	//, 20
checking	comparison
point of error, 333	pointer, 115, 116
syntax, 333	pointer to function, 115, 116
chi_squared_distribution	undefined pointer, 114, 115
probability density function, 922	unspecified pointer, 115
class, 72, 204–218	void* pointer, 115

compatible, see exception specification, compati-	typeid operator, 261
ble	virtual function call, 260
compilation	constructor, 242
separate, 16	address of, 244
compiler control line, see preprocessing directives	array of class objects and, 254
complete object, 7	converting, 247
complete object of, 7	copy, 242, 244, 245, 261-265, 406
completely defined, 208	elision, 267
component, 399	implicitly declared, 263
compound-statement, 124, 1169	implicitly defined, 264
concatenation	inaccessible, 267
macro argument, $see ##$	trivial, 264
string, $28$	default, 242, 243
condition, 125, 1169	exception handling, see exception handling,
conditions	constructors and destructors
rules for, 125	explicit call, 244
conditional-expression	implicitly called, 243
throw-expression in, 117	implicitly defined, 243
conditional-expression, 117, 1169	inheritance of, 243
conditionally-supported behavior	inheriting, 269–272
seebehavior, conditionally-supported, 1	move, 242, 244, 261–265
conflict, 11	elision, 267
conformance requirements, 5, 8	implicitly declared, 263
class templates, 5	implicitly defined, 264
classes, 5	inaccessible, 267
general, $5$	trivial, 264
library, 5	non-trivial, 243
method of description, 5	random number distribution requirement, 893
consistency	random number engine requirement, 890
linkage, 136	type of, 244
linkage specification, 166	union, $214$
type declaration, 58	unspecified argument to, 108
const, 73	constructor, conversion by, see conversion, user-
constructor and, 212, 243	defined
destructor and, 212, 249	constructor, default, see default constructor
linkage of, 55, 136	const-object
overloading and, 275	undefined change to, 144
const_cast, see cast, const	context
const_local_iterator, 720	non-deduced, 368
unordered associative containers, 720	contextually converted to bool, see conversion, con-
constant, 23, 85	textual
enumeration, 150	continue
null pointer, 80, 81	and handler, 377
constant iterator, 803	and try block, 377
constant-expression, 120, 1169	control line, see preprocessing directives
constexpr function, 140	control-line, 387, 1179
construction, 259–261	conventions, 404
dynamic cast and, 261	lexical, 16–31
member access, 259	conversion
move, 400	argument, 181
pointer to member or base, 259	array-to-pointer, 78

bool, 80	<cstddef>, 106, 114</cstddef>
boolean, 81	<pre><cstdint>, 436</cstdint></pre>
class, 246	<cstdlib>, 62, 408</cstdlib>
contextual, 77	<cstring>, 406</cstring>
derived-to-base, 287	ctor-initializer, 255, 1177
floating point, 80	<pre><cuchar>, 419</cuchar></pre>
floating to integral, 80	cv-qualifier, 73
function-to-pointer, 78	cv-qualifier, 173, 1174
implementation defined pointer integer, 101	cv-qualifier-seq, 173, 1174
implicit, 77, 246	<pre><cwchar>, 419</cwchar></pre>
implicit user-defined, 247	<pre><cwctype>, 419</cwctype></pre>
inheritance of user-defined, 249	ionotypes, iii
integer rank, 81	d-char, 27, 1165
integral, 80	d-char-sequence, 27, 1165
integral to floating, 80	DAG
lvalue-to-rvalue, 78, 1186	multiple inheritance, 221, 222
narrowing, 202	non-virtual base class, 222
overload resolution and pointer, 295	virtual base class, 221, 222
overload resolution and, 284	data race, 14
pointer, 80	data member, see member
pointer to member, 81	deadlock, 400
void*, 81	deallocation, see delete
qualification, 78–79	deallocation functions, 62
return type, 130	decay, see conversion, array to pointer; conversion,
standard, 77–82	function to pointer
static user-defined, 249	DECAY_COPY, 1117
to signed, 80	decimal-literal, 23, 1163
to unsigned, 80	decl-specifier, 134, 1171
type of, 248	decl-specifier-seq, 135, 1171
user-defined, 246–248	declaration, 32, 133–171
usual arithmetic, 84	array, 179
virtual user-defined, 249	asm, 164
conversion operator, see conversion, user defined	bit-field, 216
conversion operator, see conversion, user defined conversion rank, 288	class name, 33
•	constant pointer, 176
conversion-declarator, 248, 1177 conversion-function-id, 248, 1177	default argument, 184–187
conversion-type-id, 248, 1177	definition versus, 32
	ellipsis in function, 94, 181
conversion explicit type, see casting conversion function, see conversion, user-defined	enumerator point of, 37
	extern, 33
copy class object, see constructor, copy; assignment,	extern reference, 197
	forward, 137
copy copy constructor	forward class, 206
random number engine requirement, 890	function, 33, 181
copy elision, see constructor, copy, elision; con-	local class, 217
structor, move, elision	member, 207
copy-initialization, 192	multiple, 58
CopyInsertable into X, 706	name, 32
count	opaque enum, 33
unordered associative containers, 725	overloaded, 273
<pre>cstdarg&gt;, 181</pre>	overloaded name and friend, 237
Coudity, 101	

	150
parameter, 33, 181	namespace, 152
parentheses in, 174, 176	nested class, 216
pointer, 176	pure virtual function, 229
reference, 177	scope of class, 206
register, 135	static member, 214
static member, 33	virtual function, 228
storage class, 135	definitions, 2–5
type, 175	$\mathtt{delete}, 62, 109, 110, 252$
$ exttt{typedef}, 33$	array, 110
typedef as type, 138	destructor and, 110, 250
declaration, 133, 1170	object, $109$
declaration-seq, 133, 1170	operator, $420$
declaration-statement, 130, 1170	overloading and, 64
declaration hiding, see name hiding	type of, $252$
declarative region, 36	undefined, 110
declarator, 33, 134, 172–203	delete-expression, 109, 1168
array, 179	deleter, 513
function, 181–184	dependency-ordered before, 12
meaning of, 175–187	deprecated features, 97, 105
multidimensional array, 180	dereferencing, see also indirection
pointer, 176	derivation, see inheritance
pointer to member, 178	derived class
reference, 177	most, see most derived class
declarator, 172, 1174	derived object
declarator-id, 173, 1175	most, see most derived object
decltype-specifier, 145, 1172	derived class, 219–230
decrement operator	destruction, 259–261
overloaded, see overloading, decrement oper-	dynamic cast and, 261
ator	member access, 259
default access control, see access control, default	pointer to member or base, 259
default constructor	typeid operator, 261
random number distribution requirement, 893	virtual function call, 260
seed sequence requirement, 888	destructor, 249, 406
- · · · · · · · · · · · · · · · · · · ·	
default-initialization, 191	default, 249
defaulted, 189	exception handling, see exception handling,
default argument	constructors and destructors
overload resolution and, 284	explicit call, 250, 251
default initializers	implicit call, 250
overloading and, 275	implicitly defined, 250
deferred function, 1156	non-trivial, 249
definition, 33	program termination and, 250
alternate, 419	pure virtual, 250
class, 204, 208	union, 214
class name as type, 206	virtual, 250
constructor, 187	diagnosable rules, 5
declaration as, 134	diagnostic message, see message, diagnostic
function, 187–190	digit, 21, 1162
deleted, 189	digit-sequence, 27, 1164
explicitly-defaulted, 188	digraph, see token, alternative, 20
local class, 217	directed acyclic graph, see DAG
member function, 210	

directive, preprocessing, see preprocessing direc-	empty future object, 1151
tives	empty shared_future object, 1153
discard	empty- $declaration, 133, 1170$
random number engine requirement, 890	encoding
discard_block_engine	multibyte, 29
generation algorithm, 902	encoding-prefix, 27, 1164
state, 902	end
textual representation, 903	unordered associative containers, 726
transition algorithm, 902	end-of-file, 497
discarded-value expression, 84	endif-line, 387, 1179
discrete probability function	engine, see random number engine
bernoulli_distribution, 912	engine adaptor, see random number engine adap-
binomial_distribution, 913	tor
discrete_distribution, 925	engines with predefined parameters
geometric_distribution, 914	default_random_engine, 906
negative_binomial_distribution, 914	knuth_b, 906
poisson_distribution, 915	minstd_rand, 905
uniform_int_distribution, 910	minstd_rand0, 905
discrete_distribution	mt19937, 905
discrete probability function, 925	mt19937_64, 905
weights, 925	ranlux24, 906
distribution, see random number distribution	ranlux24_base, 905
dominance	ranlux48, 906
virtual base class, 224	$\mathtt{ranlux48\_base},905$
dot operator, see operator, class member access	entity, 32
dynamic binding, see virtual function	$\mathtt{enum},72$
dynamic initialization, 59	overloading and, 274
dynamic type, see type, dynamic	type of, $149, 150$
dynamic-exception-specification, 382, 1179	underlying type, $150$
dynamic_cast, see cast, dynamic	enum-base, $150$ , $1172$
	enum- $head, 149, 1172$
ECMA-262, 1	enum- $key$ , 149, 1172
ECMAScript, 1058, 1093	enum-name, 149, 1172
egrep, 1058	enum-specifier, 149, 1172
elaborated-type-specifier, 148, 1172	enumeration, 149, 150
elaborated type specifier, see class name, elabo-	linkage of, 56
rated	scoped, 150
elif-group, 387, 1179	unscoped, 150
elif-groups, 387, 1179	enumeration scope, 41
elision	enumeration type
copy, see constructor, copy, elision; construc-	conversion to, 100
tor, move, elision	static_cast
copy constructor, see constructor, copy, eli-	conversion to, 100
sion	enumerator
move constructor, see constructor, move, eli-	definition, 34
sion	value of, 150
ellipsis	enumerator, 150, 1172
conversion sequence, 94, 288	enumerator, 150, 1172 enumerator-definition, 150, 1172
overload resolution and, 284	
else-group, 387, 1179	enumerator-list, 150, 1172
EmplaceConstructible into X from args, 707	enum name
Empraceound of action of moin args, 101	typedef, 139

environment	nested class forward declaration, 217
program, 58	pointer to member, 178
epoch, 580	pure virtual function, 230
equal_range	scope of delete, $253$
unordered associative containers, 725	scope resolution operator, 223
equality-expression, 116, 1168	static member, 214
equivalence	subscripting, 180
template type, 314	typedef, 138
type, 138, 206	type name, 173
equivalent-key group, 719	unnamed parameter, 187
equivalent parameter declarations, 274	variable parameter list, 181
overloading and, 274	virtual function, 227, 228
erase	exception
unordered associative containers, 725	arithmetic, 83
escape-sequence, 25, 1164	undefined arithmetic, 83
escape character, see backslash	<pre><exception>, 447</exception></pre>
escape sequence	exception handling, 377–386
undefined, 26	allowing an exception, 383
Evaluation, 10	constructors and destructors, 380
evaluation	exception object, 379
order of argument, 95	constructor, 379
unspecified order of, 10, 60	destructor, 379
unspecified order of argument, 95	function try block, 378
unspecified order of function call, 95	goto, 377
example	handler, 377, 379–382, 424
array, 180	array in, 380
class definition, 209	incomplete type in, 380
	match, 380–382
const, 176	
constant pointer, 176	pointer to function in, 380
constructor, 244	rvalue reference in, 380
constructor and initialization, 254	memory, 379
declaration, 33, 183	nearest handler, 379
declarator, 173	rethrow, 379, 380
definition, 33	rethrowing, 379
delete, 252	switch, 377
derived class, 219	terminate() called, 379, 380, 383
destructor and delete, 253	throwing, 378, 379
ellipsis, 181	try block, 377
enumeration, 151	unexpected() called, 383
explicit destructor call, 251	exception object, see exception handling, excep-
explicit qualification, 223	tion object
friend, 207	exception specification, 382–385
friend function, 236	compatible, 382
function declaration, 182	incomplete type and, $382$
function definition, 187	noexcept
linkage consistency, 136	constant expression and, 382
local class, 218	virtual function and, 383
member function, 211, 236	$exception\mbox{-}declaration,\ 377,\ 1178$
nested type name, 218	$exception\mbox{-}specification,\ 382,\ 1179$
nested class, 216	exception::what message
nested class definition, 217, 240	implementation-defined, 448

exclusive-or-expression, 117, 1168	unary, $104-111$
execution agent, 1115	unary operator, 104
exit, 59, 61, 129	expression, 120, 1169
explicit-instantiation, 349, 1178	expression-list, 93, 1167
explicit-specialization, 351, 1178	expression-statement, 124, 1169
explicitly captured, 90	extended alignment, 76
explicit type conversion, see casting	extended integer type, 71
exponent-part, 26, 1164	extended signed integer type, 71
exponential_distribution	extended unsigned integer type, 71
probability density function, 916	extension-namespace-definition, 153, 1173
expression, 83–123	extern, 135
additive operators, 113	linkage of, 136
alignof, 110	extern "C", 409, 419
assignment and compound assignment, 119	extern "C++", 409, 419
bitwise AND, 116	external linkage, 55
bitwise exclusive OR, 117	extreme_value_distribution
bitwise inclusive OR, 117	probability density function, 919
cast, 95, 111–112	probability density function, 515
class member access, 96	file, source, see source file
comma, 120	final overrider, 225
conditional operator, 117	find
	unordered associative containers, 725
constant, 120	finite state machine, 1048
const cast, 102	fisher_f_distribution
decrement, 97, 105	probability density function, 923
delete, 109	floating literal, see literal, floating
dynamic cast, 97	floating-literal, 26, 1164
equality operators, 116	
function call, 93	floating-point literal, see literal, floating floating-suffix, 27, 1164
increment, 97, 105	
lambda, 88–93	floating point type, 72
left-shift-operator, 114	implementation-defined, 72
logical AND, 117	for
logical OR, 117	scope of declaration in, 128
multiplicative operators, 113	for-init-statement, 126, 1170
$\mathtt{new},106$	for-range-declaration, 126, 1170
noexcept, 110	for-range-initializer, 126, 1170
order of evaluation of, 9	formal argument, see parameter
parenthesized, 86	format specifier, 1048
pointer-to-member, 112	forwarding call wrapper, 544
pointer to member constant, 104	fractional-constant, 26, 1164
postfix, 93–104	free store, 252
primary, 85–93	freestanding implementation, 5
pseudo-destructor call, 96	free store, see also new, delete
reference, 83	friend
reinterpret cast, 101	virtual and, $228$
relational operators, 114	access specifier and, 238
right-shift-operator, 114	class access and, 236
rvalue reference, 83	inheritance and, 238
sizeof, 105	local class and, 238
static cast, 99	template and, $321$
type identification, 99	friend function

access and, $236$	function parameter, see parameter
inline, 237	function prototype, 39
linkage of, 237	function return, see return
member function and, 236	function return type, see return type
friend function	fundamental alignment, 75
nested class, 217	fundamental type
full-expression, 9	destructor and, 251
function, see also friend function; member func-	fundamental type conversion, see conversion, user
tion; inline function; virtual function, 182	defined
allocation, 63, 107	future
comparison, 399	shared state, 1146
conversion, 248	
deallocation, 64, 110, 252	gamma_distribution
definition, 34	probability density function, 917
global, 419, 422	generate
handler, 400	seed sequence requirement, 888
linkage specification overloaded, 166	generated destructor, see destructor, default
modifier, 400	generation algorithm
observer, 401	discard_block_engine, 902
operator, 296	independent_bits_engine, 903
overload resolution and, 277	linear_congruential_engine, 898
plain old, 454	mersenne_twister_engine, 899
pointer to member, 112	shuffle_order_engine, 904
replacement, 401	subtract_with_carry_engine, 900
reserved, 401	geometric_distribution
viable, 277	discrete probability function, 914
virtual member, 419, 422	global, 40
function invocation substitution, 141	global namespace, 39
function object, 539	global namespace scope, 39
binders, 550–552	global scope, 39
mem_fn, 552-553	glvalue, 74
reference_wrapper, 544	goto
type, 539	and handler, 377
wrapper, 553–557	and try block, 377
function pointer type, 72	initialization and, 130
function pointer type, 72 function try block, see exception handling, func-	grammar
tion try block	regular expression, 1093
function, overloaded, see overloading	grep, 1058
function, virtual, see virtual function	group, 387, 1179
function, virtual, see virtual function function, 187, 1175	group-part, 387, 1179
· · · · · · · · · · · · · · · · · · ·	group part, sor, 1110
function-like macro, see macro, function-like	h-char, 20, 1162
function-specifier, 137, 1171	h-char-sequence, 20, 1162
function-try-block, 377, 1178	handler, see exception handling, handler
functions	handler, 377, 1178
candidate, 344	handler-seq, 377, 1178
function argument, see argument	happens before, 13
function call, 94	hash
recursive, 95	instantiation restrictions, 558
undefined, 101	hash code, 719
function call operator	hash function, 719
overloaded, 297	

hash tables, $see$ unordered associative containers	conditional, see preprocessing directive, con-
hash_function	ditional inclusion
unordered associative containers, 723	source file, see preprocessing directives, source
hasher	file inclusion
unordered associative containers, 719	inclusive-or-expression, 117, 1168
header	incomplete, 113
C, 419, 422, 1202	increment
C library, 409	bool, $97, 105$
C++ library, 407	increment operator
name, 20–21	overloaded, see overloading, increment oper-
header-name, 20, 1162	ator
hex-quad, 17, 1161	independent_bits_engine
hexadecimal-digit, 23, 1163	generation algorithm, 903
hexadecimal-escape-sequence, 25, 1164	state, 903
hexadecimal-literal, 23, 1163	textual representation, 904
hiding, see name hiding	transition algorithm, 903
high-order bit, 6	indeterminately sequenced, 10
hosted implementation, 5	indirection, 104
•	inheritance, 219
id	init-declarator, 172, 1174
qualified, 87	init-declarator-list, 172, 1174
id-expression, 86	initialization, 59, 190–203
id-expression, 85, 1165	aggregate, 193
identifier, 21–22, 86, 134	array, 193
identifier, 21, 1162	array of class objects, 196, 254
identifier-list, 388, 1180	automatic, 130, 131
identifier-nondigit, 21, 1162	automatic object, 190
if-group, 387, 1179	base class, 255, 256
<i>if-section</i> , 387, 1179	character array, 196
ill-formed program, see program, ill-formed	character array, 196
immolation	class member, 192
self, 354	class object, see also constructor, 193, 253–
implementation	259
freestanding, 408	const, 144, 193
hosted, 408	const member, 256
implementation limits, see limits, implementation	constant, 59
implementation-defined, 419, 427, 439, 444, 446–	constructor and, 253, 254
449, 652, 973, 1026, 1199	copy, 192
implementation-defined behavior, see behavior, implemen-	default, 191
tation-defined	default constructor and, 253
implementation-dependent, 999	definition and, 134
implementation-generated, 33	
implicit object parameter, 277	direct, 192
implicitly captured, 90	dynamic, 59
implicitly-declared default constructor, see con-	explicit, 254
structor, default, 243	jump past, 130
implicit conversion, see conversion, implicit	list-initialization, 199–203
implied object argument, 277	local static, 131
implicit conversion sequences, 278	member, 255
non-static member function and, 278	member function call during, 258
inclusion	member object, 256
HICIUSIOII	order of, 59, 220

order of base class, 257	macro, 391
order of member, 257	isctype
order of virtual base class, 257	regular expression traits, 1050
overloaded assignment and, 254	iteration-statement, 126, 129, 1169
parameter, 94	, , ,
reference, 178, 196	Jessie, 247
reference member, 256	$jump\text{-}statement,\ 129,\ 1170$
run-time, 59	
static and thread, 59	key_eq
static member, 214	unordered associative containers, 723
static object, 59	key_equal
static object, 190, 191	unordered associative containers, 720
union, 196, 215	key_type
virtual base class, 265	unordered associative containers, 719
initializer	keyword, 22
base class, 187	
member, 187	label, 130
pack expansion, 259	case, 124, 126
scope of member, 258	$\mathtt{default}, 124, 126$
temporary and declarator, 245	scope of, $39, 124$
initializer, 190, 1175	labeled-statement, 124, 1169
initializer-clause, 190, 1175	lambda- $capture, 88, 1166$
initializer-list, 190, 1176	lambda- $declarator, 88, 1166$
initializer-list constructor	lambda-expression, 88, 1166
seed sequence requirement, 888	lambda-introducer, 88, 145, 1166
<pre><initializer_list>, 452</initializer_list></pre>	lattice, see DAG, subobject
injected-class-name, 204	layout
inline, 422	bit-field, 216
inline	class object, 209, 220
	layout-compatible type, 151
linkage of, 55	layout-compatible type, 70
inline function, 137	left shift
insert	undefined, 114
unordered associative containers, 724	left shift operator, 114
instantiation	lexical conventions, see conventions, lexical
explicit, 349	library
point of, 343	C standard, 399, 406, 407, 409, 1198, 1202
template implicit, 346	C++ standard, 398, 419–421, 424
instantiation units, 17	library clauses, 6
integer literal, see literal, integer	lifetime, 65
integer representation, 65	limits
integer-literal, 23, 1163	implementation, 3
integer-suffix, 23, 1163	<li>&lt;1imits&gt;, 427</li>
integer type, 71	line splicing, 16
integral type, 71	linear_congruential_engine
sizeof, 71	generation algorithm, 898
inter-thread happens before, 12	modulus, 898
internal linkage, 55	state, 898
interval boundaries	textual representation, 898
piecewise_constant_distribution, 927	transition algorithm, 898
piecewise_linear_distribution, 929	linkage, 32, 55–58
invocation	0.1 - 1

$\verb"const" and, 55$	unordered associative containers, 720
external, $55$ , $409$ , $419$	locale, 1048, 1049, 1051, 1058
implementation-defined object, 167	locale-specific behavior, see behavior, locale-specific
inline and, $55$	local class
internal, $55$	friend, $238$
no, 55, 57	member function in, 210
static and, $55$	scope of, 217
linkage specification, see specification, linkage	local scope, see block scope
linkage-specification, 164, 1173	local variable
literal, 23–31, 85	destruction of, 129, 130
base of integer, 24	logical-and-expression, 117, 1168
boolean, 29	logical-or-expression, 117, 1169
char16_t, 25	lognormal_distribution
char32_t, 25	probability density function, 921
character, 25	long
constant, 23	typedef and, 135
decimal, 24	long-long-suffix, 24, 1163
double, 27	long-suffix, 24, 1163
float, 27	lookup
floating, 26, 27	argument-dependent, 46
hexadecimal, 24	class member, 54
char, 26	class member, 49
integer, 23, 24	elaborated type specifier, 53–54
long, 24	member name, 222
<b>.</b>	name, 32, 42–55
long double, 27 multicharacter, 25	namespace aliases and, 55
	- · · · · · · · · · · · · · · · · · · ·
implementation-defined value of, 25	namespace member, 50
narrow-character, 25	qualified name, 48–53
octal, 24	template name, 331
pointer, 29	unqualified name, 42
string, 27, 28	using-directives and, 55
char16_t, 28	lookup_classname
char32_t, 28	regular expression traits, 1095
implementation-defined, 28	lookup_classname
narrow, 28	regular expression traits, 1050
type of, 28	lookup_collatename
undefined change to, 28	regular expression traits, 1050
wide, 28	low-order bit, 6
type of character, 25	lowercase, 406
type of floating point, 27	lparen, 388, 1180
type of integer, 24	lvalue, 74, 1186
unsigned, 24	lvalue reference, 72, 177
user defined, 30	
literal, 23, 1163	macro
literal type, 70	argument substitution, 391
literal-operator-id, 299, 1177	function-like, 390, 391
load_factor	arguments, 391
unordered associative containers, 726	masking, 422
local lambda expression, 90	name, 391
local variable, 38	object-like, 390, 391
local_iterator, 720	pragma operator, 397

predefined, 396	member subobject, 7
replacement, 390–395	$member-declaration,\ 208,\ 1176$
replacement list, 390	member-declarator, 208, 1176
rescanning and replacement, 393	member-declarator-list, 208, 1176
scope of definition, 393	member-specification, 207, 1176
main(), 58	members, 39
implementation-defined linkage of, 59	member data
implementation-defined parameters to, 58	static, 213
parameters to, 58	member pointer to, see pointer to member
return from, 59, 61	memory location, 6
match_results	memory model, 6–7
as sequence, 1076	memory management, see also new, delete
matched, 1048	mersenne_twister_engine
max	generation algorithm, 899
random number distribution requirement, 893	state, 899
uniform random number generator require-	textual representation, 900
ment, 889	transition algorithm, 899
max_bucket_count	g ·
unordered associative containers, 725	message diagnostic, 2, 5
	min
max_load_factor unordered associative containers, 726	
,	random number distribution requirement, 893
mean	uniform random number generator require-
normal_distribution, 920	ment, 889
poisson_distribution, 915	modification order, 11
mem-initializer, 255, 1177	most derived class, 7
mem-initializer-id, 255, 1177	most derived object, 7
mem-initializer-list, 255, 1177	bit-field, 7
member	zero size subobject, 7
class static, 62	move
enumerator, 152	class object, see constructor, move; assign-
static, 213	ment, move
template and static, 317	MoveInsertable into X, 706
member access operator	multi-pass guarantee, 806
overloaded, 298	multibyte character, see character, multibyte
member function	multicharacter literal, see literal, multicharacter
call undefined, 211	multiple threads, see threads, multiple
class, $210$	multiple inheritance, 219, 220
const, 212	virtual and, 228
constructor and, 244	multiplicative-expression, 113, 1168
destructor and, 250	$\mathtt{mutable},135$
friend, 237	mutable iterator, 803
inline, 210	mutex types, 1123
local class, 218	
nested class, 240	name, 21, 32, 86
nonstatic, 211	address of cv-qualified, 104
overload resolution and, 277	dependent, 337, 343
static, 213	elaborated
this, 212	$\mathtt{enum},148$
union, 214	global, 40
volatile, 212	length of, 21
member names, 39	macro, see macro, name
	•

point of declaration, 37	new- $expression, 106, 1167$
predefined macro, see macro, predefined	new-initializer, $106$ , $1168$
qualified, 48	new-line, 388, 1180
reserved, 418	new-placement, $106$ , $1167$
scope of, $36$	new-type-id, 106, 1167
unqualified, 42	new_handler, 63
name hiding	no linkage, 55
function, 276	noexcept-expression, 110, 1168
overloading versus, 276	noexcept-specification, 382, 1179
using-declaration and, 160	non-directive, 388, 1179
named-namespace-definition, 153, 1173	non-throwing, 384
namespace, 407, 1202	$nondigit,\ 21,\ 1162$
alias, 156	nonzero-digit, 23, 1163
definition, 152	noptr-abstract-declarator, 173, 1175
global, 419	noptr-abstract-pack-declarator, 173, 1175
member definition, 154	noptr-declarator, 172, 1174
unnamed, 154	noptr-new-declarator, 106, 1168
namespace-alias, 156, 1173	normal distributions, 920–925
namespace-alias-definition, 156, 1173	normal_distribution
namespace-body, 153, 1173	mean, 920
namespace-definition, 152, 1173 namespace-name, 152, 1173	probability density function, 920 standard deviation, 920
	· · · · · · · · · · · · · · · · · · ·
namespaces, 152–164	normative references, see references, normative
name class, see class name	notation
name hiding, 37, 42, 87, 130	syntax, 6
class definition, 206	notify_all_at_thread_exit, 1136
user-defined conversion and, 247	NTBS, 406, 1035, 1210, 1211
name space	static, 406
label, 124	NTCTS, 401
narrowing conversion, 202	NTMBS, 406
NDEBUG, 409	static, 406
negative_binomial_distribution	number
discrete probability function, 914	hex, 26
nested-name-specifier, 87, 1166	octal, $26$
nested class	${\tt numeric\_limits},427$
local class, 218	${\tt numeric\_limits}, rac{72}{}$
scope of, 216	
<new>, $439$	object, see also object model, 7, 32
$\mathtt{new},62,106,107$	byte copying and, 69
array of class objects and, 108	complete, 7
constructor and, 108	definition, 34
default constructor and, 108	delete, 110
exception and, 109	destructor static, 61
initialization and, 108	destructor and placement of, $251$
operator, $420$	linkage specification, 167
scoping and, 106	local static, $62$
storage allocation, 106	undefined deleted, 64
type of, 252	unnamed, $244$
unspecified constructor and, 108	object expression, 96
unspecified order of evaluation, 108	object model, 7
new-declarator, 106, 1167	object pointer type, 72

object representation, 69	less than or equal to, 115
object type, 7, 70	logical AND, 117
object, exception, see exception handling, excep-	logical negation, 104, 105
tion object	logical OR, 117
object-like macro, see macro, object-like	move assignment, see assignment, move
object class, see also class object	multiplication, 113
object lifetime, 65–68	multiplicative, 113
object temporary, see temporary	one's complement, 104, 105
object type, 70	overloaded, 83, 296
observable behavior, see behavior, observable	pointer to member, 112
octal-digit, 23, 1163	pragma, see macro, pragma operator
octal-escape-sequence, 25, 1164	precedence of, 9
octal-literal, 23, 1163	relational, 114
odr-used, 34	remainder, 113
one-definition rule, 34–36	scope resolution, 87, 107, 210, 219, 229
opaque-enum-declaration, 149, 1172	side effects and comma, 120
operation	side effects and logical AND, 117
atomic, 11–14	side effects and logical OR, 117
operator, 22–23, 296	$\mathtt{sizeof}, 104, 105$
*=, 119	subscripting, 93, 296
+=, 105, 119	unary, 104
-=, 119	unary minus, 104, 105
/=, 119	unary plus, 104, 105
<<=, 119	operator, 296, 1177
>>=, 119	operator delete, see also delete, 107, 110, 252
<b>%=</b> , 119	operator new, see also new, 107
&=, 119	operator overloading, see overloading, operator
^=, 119	operator!=
=, 119	random number distribution requirement, 894
additive, 113	random number engine requirement, 890
address-of, 104	operator()
assignment, 119, 406	random number distribution requirement, 893
bitwise, 116	random number engine requirement, 890
bitwise AND, 116	uniform random number generator require-
bitwise exclusive OR, 117	ment, 889
bitwise inclusive OR, 117	operator-function-id, 296, 1177
cast, 104, 173	operator<<
class member access, 96	random number distribution requirement, 894
comma, 120	random number engine requirement, 890
conditional expression, 117	operator==
copy assignment, see assignment, copy	random number distribution requirement, 894
decrement, 97, 104, 105	random number engine requirement, 890
division, 113	operator>>
equality, 116	random number distribution requirement, 894
function call, 93, 296	random number engine requirement, 891
greater than, 115	operator, see delete
greater than or equal to, 115	operator left shift, see left shift operator
increment, 97, 104, 105	operator right shift, see right shift operator
indirection, 104	operator use
inequality, 116	scope resolution, 214
less than, 115	~~····································

optimization of temporary, see elimination of tem-	scoping ambiguity, 223
porary	template, 329
order of evaluation in expression, see expression,	template name, 331
order of evaluation of	viable functions, 284–296
ordering	subscripting operator, 297
function template partial, 329	unary operator, 296
order of execution	user-defined literal, 299
base class constructor, 244	using directive and, 163
base class destructor, 250	using-declaration and, 160
constructor and static objects, 255	overloads
constructor and array, 253	floating point, 886
destructor, 250	overrider
destructor and array, 250	final, 225
member constructor, 244	own, 513
member destructor, 250	,
original-namespace-definition, 153, 1173	pair
original-namespace-name, 152, 1173	tuple interface to, 476
over-aligned type, 76	param
overflow, 83	random number distribution requirement, 893
undefined, 83	seed sequence requirement, 888
overloaded function, see overloading	param_type
overloaded operator, see overloading, operator	random number distribution requirement, 893
overloadedfunction	parameter, 3
address of, 294	catch clause, 3
overloaded function	function, 3
address of, 105	function-like macro, 3
overloaded operator	reference, 177
inheritance of, 296	scope of, 38
overloading, 181, 206, 273–302, 327	template, 3, 33
access control and, 276	void, 181
address of overloaded function, 294	parameter declaration, 33
argument lists, 277–284	parameter-declaration, 181, 1175
assignment operator, 297	parameter-declaration-clause, 181, 1175
binary operator, 297	parameter-declaration-list, 181, 1175
built-in operators and, 299	parameterized type, see template
candidate functions, 277–284	parameters
declaration matching, 275	macro, 391
declaration matching, 273 declarations, 273	parameters-and-qualifiers, 172, 1174
example of, 273	parameter list
function call operator, 297	variable, 94, 181
member access operator, 298	period, 406
operator, 296–299	phases of translation, see translation, phases
prohibited, 273	piecewise construction, 478
resolution, 276–294	piecewise_constant_distribution
best viable function, 284–296	interval boundaries, 927
•	probability density function, 927
contexts, 277	weights, 927
function call syntax, 278–280	piecewise_linear_distribution
function template, 374	interval boundaries, 929
implicit conversions and, 286–294	probability density function, 929
initialization, 282–284	weights at boundaries, 929
operators, 280	weights at boundaries, 323

placement syntax	source-file inclusion, 389
$\mathtt{new},108$	preprocessing-file, 387, 1179
pm-expression, 112, 1168	preprocessing-op-or-punc, 23, 1162
POD class, 205	preprocessing-token, 19, 1161
POD struct, 205	primary equivalence class, 1049
POD union, 205	primary-expression, 85, 1165
POF, 454	private, see access control, private
point of declaration, 37	probability density function
pointer, see also void*	cauchy_distribution, 922
safely-derived, 64–65	chi_squared_distribution, 922
to traceable object, 424	exponential_distribution, 916
to traceable object, 64	extreme_value_distribution, 919
zero, 80	fisher_f_distribution, 923
pointer literal, see literal, pointer	gamma_distribution, 917
pointer, integer representation of safely-derived,	lognormal_distribution, 921
65	normal_distribution, 920
pointer-literal, 29, 1165	piecewise_constant_distribution, 92'
pointer to member, 72, 112	piecewise_linear_distribution, 929
Poisson distributions, 915–920	student_t_distribution, 924
poisson_distribution	uniform_real_distribution, 911
discrete probability function, 915	weibull_distribution, 918
mean, 915	program, 55
POSIX, 1	ill-formed, 3
extended regular expressions, 1058	start, 58–61
regular expressions, 1058	termination, 61–62
postfix-expression, 93, 1167	well-formed, 5, 8
postfix ++ and	program execution, 8–11
overloading, 298	abstract machine, 8
postfix ++, 97	as-if rule, see as-if rule
postfix, 97	promotion
potential scope, 36	bool to int, 80
potentially evaluated, 34	floating point, 80
pp-number, 21, 1162	integral, 79
pp-tokens, 388, 1180	protected, see access control, protected
precedence of operator, see operator, precedence	protection, see access control, 424
of	prvalue, 74
prefix	pseudo-destructor-name, 96
L, 25, 28	pseudo-destructor-name, 93, 1167
prefix ++ and	ptr-abstract-declarator, 173, 1175
overloading, 298	ptr-declarator, 172, 1174
prefix ++, 105	ptr-operator, 173, 1174
prefix, 105	ptrdiff_t, 114
preprocessing directive, 387	implementation defined type of, 114
conditional inclusion, 388	public, see access control, public
preprocessing directives, 387–397	punctuator, 22–23
error, 395	pure-specifier, 208, 1176
header inclusion, 389	a shan 91 1169
line control, 395	q-char, 21, 1162
macro replacement, see macro, replacement	q-char-sequence, 21, 1162
null, 396	qualification
pragma, 396	explicit, 48

qualified-id, 87, 1166	random_device
qualified-namespace-specifier, 156, 1173	implementation leeway, 906
	raw string literal, 28
r-char, 27, 1165	raw-string, 27, 1164
r-char-sequence, 27, 1164	reaching scope, 90
random number distribution	ready, 1076, 1147
bernoulli_distribution, 912	redefinition
binomial_distribution, 913	typedef, 138
${\tt chi\_squared\_distribution}, 922$	ref-qualifier, 173, 1174
discrete_distribution, 925	reference, 72
exponential_distribution, 916	assignment to, 119
extreme_value_distribution, 919	call by, 94
fisher_f_distribution, 923	lvalue, 72
gamma_distribution, 917	null, 178
geometric_distribution, 914	rvalue, 72
lognormal_distribution, 921	
negative_binomial_distribution, 914	sizeof, 105
normal_distribution, 920	reference collapsing, 178
piecewise_constant_distribution, 927	reference-compatible, 197
piecewise_linear_distribution, 929	reference-related, 197
	references
poisson_distribution, 915	normative, 1
requirements, 892–895	regex_iterator
student_t_distribution, 924	end-of-sequence, 1088
uniform_int_distribution, 910	regex_token_iterator
uniform_real_distribution, 911	end-of-sequence, 1090
random number distributions	regex_traits
Bernoulli, 912–915	specializations, 1062
normal, $920-925$	region
Poisson, 915–920	declarative, 32, 36
sampling, $925-930$	register, $135$
uniform, 910–912	regular expression, 1048–1095
random number engine	grammar, 1093
${ t linear\_congruential\_engine},898$	matched, 1048
mersenne_twister_engine, 899	requirements, 1049
requirements, 889–891	regular expression traits, 1093
${\tt subtract\_with\_carry\_engine},900$	char_class_type, 1049
with predefined parameters, 905–906	isctype, $1050$
random number engine adaptor	lookup_classname, 1095
discard_block_engine, 902	lookup_classname, 1050
independent_bits_engine, 903	lookup_collatename, 1050
shuffle_order_engine, 904	requirements, 1049, 1062
with predefined parameters, 905–906	transform, 1094
random number generation, 886–930	transform, 1050
distributions, 910–930	transform_primary, 1095
engines, 897–905	transform_primary, 1050
predefined engines and adaptors, 905–906	translate, 1094
requirements, 887–895	
synopsis, 895–897	translate, 1049
utilities, 907–910	translate_nocase, 1094
random number generator, see uniform random	translate_nocase, 1050
number generator, see uniform random	rehash
number generator	unordered associative containers, 727

reinterpret_cast, see cast, reinterpret	operator overloading, 296
relational-expression, 115, 1168	overloading, 296
relaxed pointer safety, 65	pointer to bit-field, 216
release sequence, 12	reference, 178
remainder operator, see remainder operator	register, $135$
replacement	static, 136
macro, see macro, replacement	static member local class, 214
replacement-list, 388, 1180	union, 214
representation	result_type
object, 69	entity characterization based on, 886
value, 69	result_type
requirements, 402	random number distribution requirement, 893
Allocator, 413	seed sequence requirement, 888
container, 702, 719, 730, 731, 1076	uniform random number generator require-
not required for unordered associated con-	ment, 889
tainers, 718	rethrow, see exception handling, rethrow
CopyAssignable, 410	return, 129, 130
CopyConstructible, 410	and handler, 377
DefaultConstructible, 410	and try block, 377
Destructible, 410	constructor and, 130
EqualityComparable, 410	reference and, 197
Hash, $413$	return statement, see return
iterator, 803	return type, 182
LessThanComparable, 410	overloading and, 273
MoveAssignable, 410	right shift
MoveConstructible, 410	implementation defined, 114
NullablePointer, 413	right shift operator, 114
numeric type, 874	rounding, 80
random number distribution, 892–895	rvalue, 74
random number engine, 889–891	lvalue conversion to, see conversion, lvalue to
regular expression traits, 1049, 1062	rvalue
seed sequence, 887–888	lvalue conversion to, 1186
sequence, 1076	rvalue reference, 72, 177
uniform random number generator, 888–889	110100 101010100, 12, 111
unordered associative container, 719	s-char, 27, 1164
reraise, see exception handling, rethrow	s-char-sequence, 27, 1164
rescanning and replacement, see macro, rescan-	safely-derived pointer, 64
ning and replacement	integer representation, 65
reserved identifier, 22	sampling distributions, 925–930
reset, 513	scalar type, 70
reset	scope, 1, 32, 36–42, 134
random number distribution requirement, 893	anonymous union at namespace, 215
resolution, see overloading, resolution	block, 38
restriction, 421, 422, 424	class, 40
address of bit-field, 216	declarations and, 36–38
anonymous union, 215	destructor and exit from, 129
bit-field, 216	enumeration, 41
constructor, 243, 244	exception declaration, 38
destructor, 249	function, 39
extern, 136	function prototype, 39
local class, 218	global, 39
10001 01000, 210	<b>∀</b> ,

global namespace, 39	simple call wrapper, 544
$iteration\text{-}statement,\ 127$	simple-declaration, 133, 1170
macro definition, see macro, scope of defini-	simple-escape-sequence, $25$ , $1164$
tion	$simple-template-id,\ 307,\ 1178$
namespace, 39	simple-type-specifier, 145, 1172
name lookup and, 42–55	size
overloading and, 275	seed sequence requirement, 888
potential, 36	size_t, 106
selection-statement, 125	smart pointers, 523–539
template parameter, 41	source file, 16, 409, 419
scope name hiding and, 42	source file character, see character, source file
scope resolution operator, 48	space
seed	white, 19
random number engine requirement, 890	specialization
seed sequence, 887	class template, 308
requirements, 887–888	class template partial, 323
selection-statement, 125, 1169	template, 345
semantics	template explicit, 351
class member, 96	special member function, see constructor, destruction, destruction, see constructor, destruction, destruction
separate compilation, see compilation, separate	tor, inline function, user-defined conver
separate translation, see compilation, separate	sion, virtual function
, , , ,	specification
sequence	linkage, 164–167
ambiguous conversion, 287	
implicit conversion, 286	extern, 164
standard conversion, 77	implementation-defined, 165
sequence constructor	nesting, 165
seed sequence requirement, 888	template argument, 357
Sequenced before, 10	specifications
sequencing operator, see comma operator	C standard library exception, 424
setlocale, 406	C++, 424
shared state, see future, shared state	implementation-defined exception, 424
shift-expression, 114, 1168	specifier, 134–149
shift operator, see left shift operator, right shift op-	friend, 424
erator	constexpr, 139
short	constructor, 140, 141
typedef and, 135	function, 140
shuffle_order_engine	cv-qualifier, 144
generation algorithm, 904	declaration, 134
state, 904	explicit, 138
textual representation, 905	$\mathtt{friend}, 139$
transition algorithm, 904	function, 137
side effects, 8, 10–14, 117, 124, 245, 257, 267, 393,	inline, 137
424	missing storage class, 136
visible, 13	$\mathtt{static},135$
visible sequence of, 13	storage class, 135
sign,  27,  1164	type, see type specifier
signal, 8	typedef, 138
signature, 3, 4	virtual, 137
signed	specifier access, see access specifier
typedef and, 135	stable algorithm, 401, 423
signed integer type, 71	stack unwinding

see exception handling, constructors and de-	statement-seq, 124, 1169
structors, 380	$\mathtt{static},135$
standard	destruction of local, 131
structure of, 5–6	linkage of, 55, 136
standard deviation	overloading and, 273
normal_distribution, 920	static initialization, 59
standard-layout types, 70	static storage duration, 62
standard-layout class, 205	static type, see type, static
standard-layout struct, 205	static_assert, 134
standard-layout union, 205	static_assert-declaration, 133, 1170
standard integer type, 71	static_cast, see cast, static
standard signed integer type, 71	<pre><stddef.h>, 25, 28</stddef.h></pre>
standard unsigned integer type, 71	<stdexcept>, 456</stdexcept>
start	storage-class-specifier, 135, 1171
program, 59	storage class, 32
startup	storage duration, 62–65
program, 409, 420	automatic, 62
state	class member, 65
discard_block_engine, 902	dynamic, 62–65, 106
independent_bits_engine, 903	local object, 62
linear_congruential_engine, 898	register, 62
mersenne_twister_engine, 899	static, 62
object, 400	thread, 62
· ·	
shuffle_order_engine, 904	storage management, see new, delete
subtract_with_carry_engine, 900	stream
statement, 124–132	arbitrary-positional, 399
continue in for, 128	repositional, 401
break, 129	streambuf
compound, 124	implementation-defined, 961
continue, 129	strict pointer safety, 65
declaration, 130	string
declaration in if, 125	distinct, 28
declaration in switch, 125	null-terminated byte, 406
declaration in for, 128	null-terminated character type, 401
declaration in switch, 126	null-terminated multibyte, 406
declaration in while, 127	$\mathtt{sizeof}, {\color{red} 29}$
do, 126, 127	type of, $28$
empty, $124$	string literal, see literal, string
expression, 124	string-literal, 27, 1164
for, 126, 128	stringize, $see \#$
goto, 124, 129, 130	struct
if, 125, 126	standard-layout, 205
iteration, 126–129	struct
jump, 129	class versus, $204$
labeled, 124	structure, 204
null, 124	structure tag, see class name
for, 128	student_t_distribution
selection, 125–126	probability density function, 924
switch, 125, 126, 129	sub-expression, 1049
while, 126, 127	subobject, see also object model, 7
statement, 124, 1169	subscripting operator
	1 0 1

overloaded, 297	linkage of, $303$
subsequence rule	template parameter scope, 41
overloading, 292	temporary, 244
subtract_with_carry_engine	constructor for, 245
carry, 900	destruction of, 245
generation algorithm, 900	destructor for, 245
state, 900	elimination of, 244, 267
textual representation, 901	implementation-defined generation of, 244
transition algorithm, 900	order of destruction of, 245
subtraction	terminate(), $385$ , $386$
implementation defined pointer, 114	called, 379, 380, 383, 385
subtraction operator, 113	termination
suffix	program, 59, 62
E, 27	terminology
e, 27	pointer, 72
F, 27	text-line, 387, 1179
f, 27	textual representation
L, 24, 27	discard_block_engine, 903
1, 24, 27	independent_bits_engine, 904
U, 24	shuffle_order_engine, 905
u, 24	subtract_with_carry_engine, 901
summary	this, $85, 212$
x C++ 2003, 1191	type of, 212
compatibility with ISO C, 1183	this pointer, see this
swappable, 412	thread, 11
swappable with, 411	thread of execution, 11
switch	thread storage duration, 62
and handler, 377	thread, blocked, 399
and try block, 377	$thread_local, 135$
synchronize with, 12	threads
synonym, 156	m multiple,  11-14
type name as, 138	throw, $377$
syntax	throw-expression, 377, 1178
class member, 96	throwing, see exception handling, throwing
	timed mutex types, 1126
target object, 543	token, 20
template, 303–376	alternative, 20
definition of, 303	preprocessing, 19
function, 356	token, 20, 1162
member function, 316	traceable pointer object, 64, 424
primary, 323	trailing-return-type, 172, 1174
template, $303$	trailing-type-specifier, 143, 1171
template parameter, 33	trailing-type-specifier-seq, 143, 1171
template-argument, 307, 1178	traits, 401
template-argument-list, 307, 1178	transfer ownership, 513
template-declaration, 303, 1177	transform
template-id, 307, 1178	regular expression traits, 1094
template-name, 307, 1178	transform
template-parameter, 304, 1178	regular expression traits, 1050
template-parameter-list, 303, 1178	transform_primaryl
template name	regular expression traits, 1095

transform_primary	class and, $204$
regular expression traits, 1094	compound, 72
transform_primaryl	$\mathtt{const},143$
regular expression traits, 1050	destination, 192
TransformationTrait, 559	double, 72
transition algorithm	dynamic, 2
discard_block_engine, 902	enumerated, 72, 404, 405
independent_bits_engine, 903	enumeration underlying, 151
linear_congruential_engine, 898	example of incomplete, 69
mersenne_twister_engine, 899	extended integer, 71
shuffle_order_engine, 904	extended signed integer, 71
subtract_with_carry_engine, 900	extended unsigned integer, 71
translate	float, $72$
regular expression traits, 1094	floating point, 71
translate	function, 72, 181, 182
regular expression traits, 1049	fundamental, 71
translate_nocase	sizeof, 71
regular expression traits, 1094	incomplete, 34, 37, 69, 78, 93–97, 99, 104–106,
translate_nocase	110, 119, 219
regular expression traits, 1050	int, 71
translation	integral, 71
phases, 16–17	long, 71
separate, see compilation, separate	long double, 72
translation unit, 16	long long, 71
translation units, 55	multi-level mixed pointer and pointer to mem-
translation-unit, 55, 1165	ber, 79
translation unit, 55	multi-level pointer to member, 79
name and, 32	over-aligned, 76
trigraph sequence, 16, 18	POD, 70
trivial types, 70	pointer, 72
trivially copyable class, 204	polymorphic, 225
trivially copyable types, 70	$\mathtt{short}, 71$
trivial class, 204	signed char, $71$
trivial class type, 108	signed integer, 71
trivial type, 108	standard integer, 71
truncation, 80	standard signed integer, 71
try, 377	standard unsigned integer, 71
try block, see exception handling, try block	static, 4
try-block, 377, 1178	trivially copyable, 69
tuple	underlying wchar_t, 71
and pair, $476$	unsigned, 71
type, 32, 69–74	unsigned char, 71
arithmetic, 72	unsigned int, 71
array, 72, 182	unsigned long, 71
bitmask, 405	unsigned long long, 71
Boolean, 71	unsigned short, 71
char, 71	unsigned integer, 71
char16_t, 71	void, 72
char32_t, 71	volatile, $143$
character, 71	wchar_t, 71
character container, 399	type generator, see template
,	v 1 O , 1

type specifier	unsigned, 145
auto, 148	void, 145
const, 144	volatile, 145
elaborated, 148	
simple, 145	ud-suffix, 30, 1165
volatile, 144	unary function, 544
type-id, 173, 1175	unary operator
type-id-list, 382, 1179	overloaded, 296
type-name, 145, 1172	unary-expression, 104, 1167
type-parameter, 304, 1178	unary-operator, 104, 1167
type-specifier	UnaryTypeTrait, 558
bool, 145	unary operator
wchar_t, 145	interpretation of, 296
type-specifier, 143, 1171	unblock, 401
type-specifier-seq, 143, 1171	uncaught_exception(), 386
type_info, 99	undefined, 401, 419, 421, 935, 937, 940–943, 947,
typedef	951, 975
function, 182	undefined behavior, see behavior, undefined, 831
typedef	underlying type, 71
overloading and, 274	unevaluated operand, 84
typedef-name, 138, 1171	unexpected(), 384, 386
typeid, 99	called, 383
construction and, 261	Unicode required set, 397
destruction and, 261	uniform distributions, 910–912
<pre><typeinfo>, 445</typeinfo></pre>	uniform random number generator
typename, 148	requirements, 888–889
typename-specifier, 332, 1178	uniform_int_distribution
types	discrete probability function, 910
implementation-defined, 404	uniform_real_distribution
implementation-defined exception, 424	probability density function, 911
type checking	union
argument, 94	standard-layout, 205
type conversion, explicit, see casting	union, 72, 214
type name, 173	class versus, 204
nested, 218	anonymous, 215
scope of, 218	global anonymous, 215
type pun, 102	unique pointer, 513
type specifier	unit
auto, 145	translation, 409, 419
char, 145	universal character name, 16
char16_t, 145	universal-character-name, 18, 1161
char32_t, 145	unnamed-namespace-definition, 153, 1173
decltype, 145, 147	unordered associative containers, 718–792
double, 145	begin, 726
elaborated, 53	bucket, 726
enum, 148	bucket_count, 725
float, 145	bucket_size, 726
int, 145	cbegin, 726
long, 145	cend, 726
short, 145	clear, $725$
signed, 145	complexity, 718
0	- **

${\tt const\_local\_iterator}, 720$	user-defined-literal, $30, 1165$
$\mathtt{count},725$	user-defined- $string$ - $literal, 30, 1165$
end, $726$	user-provided, 189
equal_range, 725	Uses-allocator construction, 505
equality function, 719	using-declaration, 156–161
equivalent keys, 719, 783, 789	using-declaration, 156, 1173
$\frac{1}{2}$ erase, $\frac{725}{2}$	using-directive, 161–164
exception safety, 727, 728	using-directive, 161, 1173
find, 725	usual arithmetic conversions, see conversion, usual
hash function, 719	arithmetic
hash_function, 723	
hasher, 719	valid, 36
insert, 724	valid but unspecified state, 401
iterator invalidation, 727	value, 69
	call by, 94
iterators, 727	null member pointer, 81
key_eq, 723	null pointer, 80
key_equal, 720	undefined unrepresentable integral, 80
key_type, 719	-
lack of comparison operators, 718	value category, 74
load_factor, 726	value computation, 10–11, 13, 14, 97, 108, 117,
$local\_iterator, 720$	119, 120, 245
max_bucket_count, 725	value representation, 69
$ exttt{max\_load\_factor}, 726$	value-initialization, 191
rehash, 727	ValueSwappable, 412
requirements, 718, 719, 727, 728	variable, 32
unique keys, 719, 778, 786	indeterminate uninitialized, 191
unordered_map	virt-specifier, 208, 1176
element access, 782	virt-specifier-seq, 208, 1176
unique keys, 778	virtual base class, 221
unordered_multimap	virtual function, 225–229
equivalent keys, 783	pure, 229, 230
unordered_multiset	virtual function call, 229
equivalent keys, 789	constructor and, 260
unordered_set	destructor and, 260
unique keys, 786	undefined pure, 230
unqualified-id, 85, 1166	visibility, 42
unsequenced, 10	visible, 42
unsigned	void*
0	type, 73
typedef and, 135	void&, 177
unsigned-suffix, 24, 1163	volatile, 73
unsigned integer type, 71	constructor and, 212, 243
unspecified, 440, 441, 445, 861, 1025, 1206, 1208	
unspecified behavior, see behavior, unspecified, 941	destructor and, 212, 249
unwinding	implementation-defined, 145
stack, 380	overloading and, 275
uppercase, 406, 419	waiting function 1146
user-defined literal, see literal, user defined	waiting function, 1146
overloaded, 299	wchar_t, 25, 28, 641
user-defined-character-literal, 30, 1165	implementation-defined, 71
user-defined-floating-literal, 30, 1165	weak result type, 543
user-defined-integer-literal, 30, 1165	weibull_distribution

© ISO/IEC N3337

```
probability density function, 918
weights
    discrete_distribution, 925
    piecewise_constant_distribution, 927
weights at boundaries
    piecewise_linear_distribution, 929
well-formed program, see program, well-formed
white space, 20
wide-character, 25
X(X&), see copy constructor
xvalue, 74
zero
    division by undefined, 83
    remainder undefined, 83
    undefined division by, 113
zero-initialization, 191
```

© ISO/IEC N3337

## 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).

$abstract-declarator,\ 173,\ 1175$	class-specifier, 204, 1176
abstract-pack-declarator, 173, 1175	class-virt-specifier, 204, 1176
access-specifier, 219, 1177	compound-statement, 124, 1169
additive-expression, 113, 1168	condition, 125, 1169
alias-declaration, 133, 1170	conditional-expression, 117, 1169
alignment-specifier, 167, 1173	constant-expression, 120, 1169
and-expression, 116, 1168	control-line, 387, 1179
asm-definition, 164, 1173	conversion-declarator, 248, 1177
assignment-expression, 119, 1169	conversion-function-id, 248, 1177
assignment-operator, 119, 1169	conversion-type-id, 248, 1177
attribute, 167, 1173	ctor-initializer, 255, 1177
attribute-argument-clause, 168, 1174	cv-qualifier, 173, 1174
attribute-declaration, 133, 1170	cv-qualifier-seq, 173, 1174
attribute-list, 167, 1173	
attribute-namespace, 168, 1174	d-char, 27, 1165
attribute-scoped-token, 168, 1174	d-char-sequence, 27, 1165
attribute-specifier, 167, 1173	decimal-literal, 23, 1163
attribute-specifier-seq, 167, 1173	decl-specifier, 134, 1171
attribute-token, 167, 1174	decl-specifier-seq, 135, 1171
	declaration, 133, 1170
balanced-token, 168, 1174	declaration-seq, 133, 1170
balanced-token-seq, 168, 1174	declaration-statement, 130, 1170
base-clause, 219, 1176	declarator, 172, 1174
base-specifier, 219, 1177	declarator- $id$ , 173, 1175
base-specifier-list, 219, 1177	decltype-specifier, 145, 1172
base-type-specifier, 219, 1177	delete-expression, 109, 1168
block-declaration, 133, 1170	digit, 21, 1162
boolean-literal, 29, 1165	digit-sequence, 27, 1164
brace-or-equal-initializer, 190, 1175 braced-init-list, 190, 1176	dynamic-exception-specification, 382, 1179
074004-11111-1130, 130, 1170	elaborated-type-specifier, 148, 1172
c-char, 25, 1163	elif-group, 387, 1179
c-char-sequence, 25, 1163	elif-groups, 387, 1179
capture, 88, 1166	else-group, 387, 1179
capture-default, 88, 1166	empty-declaration, 133, 1170
capture-list, 88, 1166	encoding-prefix, 27, 1164
cast-expression, 111, 1168	endif-line, 387, 1179
character-literal, 25, 1163	enum-base, 150, 1172
class-head, 204, 1176	enum-head, 149, 1172
class-head-name, 204, 1176	enum-key, 149, 1172
class-key, 204, 1176	enum-name, 149, 1172
class-name, 204, 1176	enum-specifier, 149, 1172
class-or-decltype, 219, 1177	enumerator, 150, 1172
couss or accompte, 213, 1111	Chameer and 1, 100, 1112

enumerator-definition, 150, 1172	integer-suffix, 23, 1163
enumerator-list, 150, 1172	iteration-statement, 126, 1169
equality-expression, 116, 1168	
escape-sequence, 25, 1164	$jump\text{-}statement,\ 129,\ 1170$
exception-declaration, 377, 1178	111111111111111111111111111111111111111
exception-specification, 382, 1179	labeled-statement, 124, 1169
exclusive-or-expression, 117, 1168	lambda-capture, 88, 1166
explicit-instantiation, 349, 1178	lambda-declarator, 88, 1166
explicit-specialization, 351, 1178	lambda-expression, 88, 1166
exponent-part, 26, 1164	lambda-introducer, 88, 1166
expression, 120, 1169	linkage-specification, 164, 1173
expression-list, 93, 1167	literal, 23, 1163
expression-statement, 124, 1169	literal-operator-id, 299, 1177
extension-namespace-definition, 153, 1173	logical-and-expression, 117, 1168
	logical-or-expression, 117, 1169
floating-literal, 26, 1164	long-long-suffix, 24, 1163
floating-suffix, 27, 1164	long-suffix, 24, 1163
for-init-statement, 126, 1170	lparen, 388, 1180
for-range-declaration, 126, 1170	mem-initializer, 255, 1177
for-range-initializer, 126, 1170	mem-initializer-id, 255, 1177
fractional-constant, 26, 1164	mem-initializer-list, 255, 1177
function-definition, 187, 1175	member-declaration, 208, 1176
function-specifier, 137, 1171	member-accturation, 208, 1176 $member$ -declarator, 208, 1176
function-try-block, 377, 1178	member-declarator, 208, 1176  member-declarator-list, 208, 1176
207 1170	member-specification, 207, 1176
group, 387, 1179	multiplicative-expression, 113, 1168
group-part, 387, 1179	munipulative expression, 110, 1100
h-char, 20, 1162	$named-name space-definition,\ 153,\ 1173$
h-char-sequence, 20, 1162	$name space-alias,\ 156,\ 1173$
handler, 377, 1178	$name space-alias-definition,\ 156,\ 1173$
handler-seq, 377, 1178	name space-body, 153, 1173
header-name, 20, 1162	$name space-definition,\ 152,\ 1173$
hex-quad, 17, 1161	$name space-name,\ 152,\ 1173$
hexadecimal-digit, 23, 1163	nested- $name$ - $specifier, 87, 1166$
hexadecimal-escape-sequence, 25, 1164	new- $declarator$ , $106$ , $1167$
hexadecimal-literal, 23, 1163	new- $expression, 106, 1167$
	new-initializer, $106$ , $1168$
id-expression, 85, 1165	new-line, 388, 1180
identifier, 21, 1162	new-placement, $106$ , $1167$
identifier-list, 388, 1180	new- $type$ - $id$ , $106$ , $1167$
identifier-nondigit, 21, 1162	$no except\mbox{-}expression,110,1168$
if-group, 387, 1179	$no except\mbox{-}specification,\ 382,\ 1179$
if-section, 387, 1179	$non\text{-}directive,\ 388,\ 1179$
inclusive-or-expression, 117, 1168	$nondigit,\ 21,\ 1162$
init-declarator, 172, 1174	nonzero-digit, 23, 1163
init-declarator-list, 172, 1174	$noptr-abstract-declarator,\ 173,\ 1175$
initializer, 190, 1175	$noptr-abstract-pack-declarator,\ 173,\ 1175$
initializer-clause, 190, 1175	$noptr\mbox{-}declarator,172,1174$
initializer-list, 190, 1176	$noptr-new-declarator,\ 106,\ 1168$
integer-literal, 23, 1163	
	octal- $digit$ , 23, 1163

octal-escape-sequence, 25, 1164	$storage\text{-}class\text{-}specifier,\ 135,\ 1171$
octal-literal, 23, 1163	string-literal, 27, 1164
opaque-enum-declaration, 149, 1172	
operator, 296, 1177	$template-argument,\ 307,\ 1178$
operator-function-id, 296, 1177	template-argument-list, 307, 1178
original-namespace-definition, 153, 1173	$template\text{-}declaration,\ 303,\ 1177$
original-namespace-name, 152, 1173	template-id, 307, 1178
,,,	template-name, 307, 1178
parameter-declaration, 181, 1175	template-parameter, $304$ , $1178$
parameter-declaration-clause, 181, 1175	template-parameter-list, 303, 1178
parameter-declaration-list, 181, 1175	text-line, 387, 1179
parameters-and-qualifiers, 172, 1174	throw-expression, 377, 1178
pm-expression, 112, 1168	token, 20, 1162
pointer-literal, 29, 1165	trailing-return-type, 172, 1174
postfix-expression, 93, 1167	trailing-type-specifier, 143, 1171
pp-number, 21, 1162	trailing-type-specifier-seq, 143, 1171
pp-tokens, 388, 1180	translation-unit, 55, 1165
preprocessing-file, 387, 1179	try-block, 377, 1178
preprocessing-op-or-punc, 23, 1162	type-id, 173, 1175
preprocessing-token, 19, 1161	type-id-list, 382, 1179
primary-expression, 85, 1165	type-name, 145, 1172
pseudo-destructor-name, 93, 1167	type-name, 140, 1172 type-parameter, 304, 1178
-	
ptr-abstract-declarator, 173, 1175	type-specifier, 143, 1171
ptr-declarator, 172, 1174	type-specifier-seq, 143, 1171
ptr-operator, 173, 1174	typedef-name, 138, 1171
pure-specifier, 208, 1176	typename-specifier, 332, 1178
q-char, 21, 1162	ud-suffix, $30$ , $1165$
q-char-sequence, 21, 1162	unary-expression, 104, 1167
qualified-id, 87, 1166	unary-operator, 104, 1167
qualified-namespace-specifier, 156, 1173	universal-character-name, 18, 1161
qualifica mantespace specifici, 100, 1110	unnamed-namespace-definition, 153, 1173
r-char, 27, 1165	unqualified-id, 85, 1166
r-char-sequence, 27, 1164	unsigned-suffix, 24, 1163
raw-string, 27, 1164	user-defined-character-literal, 30, 1165
ref-qualifier, 173, 1174	user-defined-floating-literal, 30, 1165
relational-expression, 115, 1168	user-defined-integer-literal, 30, 1165
replacement-list, 388, 1180	· · · · · · · · · · · · · · · · · · ·
replacement toot, 500, 1100	user-defined-literal, 30, 1165
s-char, 27, 1164	user-defined-string-literal, 30, 1165
s-char-sequence, 27, 1164	using-declaration, 156, 1173
selection-statement, 125, 1169	using-directive, 161, 1173
shift-expression, 114, 1168	
sign, 27, 1164	virt-specifier, 208, 1176
simple-declaration, 133, 1170	virt-specifier-seq, 208, 1176
· · · · · · · · · · · · · · · · · · ·	
simple-escape-sequence, 25, 1164	
$simple-template-id,\ 307,\ 1178$	
simple-template-id, 307, 1178 simple-type-specifier, 145, 1172	
simple-template-id, 307, 1178 simple-type-specifier, 145, 1172 statement, 124, 1169	
simple-template-id, 307, 1178 simple-type-specifier, 145, 1172	

© ISO/IEC N3337

## Index of library names

```
<scoped_allocator>, 593
                                                          operator==, 509
                                                      allocator_arg, 504
_{\rm Exit}, 438
_{\tt lignas_is\_defined},\,454
                                                      allocator_arg_t, 504
__bool_true_false_are_defined, 453, 454
                                                      allocator_traits, 505
                                                          allocate, 507
_1, 552
                                                          const_pointer, 506
                                                          const_void_pointer, 506
а
                                                          constructor, 507
    cauchy_distribution, 923
                                                          deallocate, 507
    extreme_value_distribution, 919
                                                          destructor, 507
    uniform_int_distribution, 911
                                                          difference_type, 506
    uniform_real_distribution, 912
                                                          max_size, 507
    weibull_distribution, 919
                                                          {\tt pointer},\, {\tt 506}
abort, 62, 129, 408, 438, 444, 449
                                                          propagate_on_container_copy_assignment,
abs, 944, 956
    complex, 883
                                                          propagate_on_container_move_assignment,
accumulate, 953
acos, 944, 956
                                                          propagate_on_container_swap, 507
    complex, 884
                                                          rebind_alloc, 507
acosh, 956
                                                          select_on_container_copy_construction,
    complex, 884
                                                               507
address
                                                          size_type, 506
    allocator, 508, 509
                                                          void_pointer, 506
addressof, 511
adjacent_difference, 954
                                                          gamma_distribution, 918
adjacent_find, 848
                                                      always_noconv
advance, 814
                                                          codecvt, 667
<algorithm>, 836
                                                      any
align, 504
                                                          bitset, 496
all
                                                      any_of, 846
    bitset, 496
                                                      append
all_of, 846
                                                          basic_string, 622, 623
allocate
                                                      apply
    allocator, 509
                                                          valarray, 941
    allocator_traits, 507
                                                      arg, 886
    scoped_allocator_adaptor, 596
                                                          complex, 883
allocate_shared, 530
                                                      <array>, 728
allocator, 1081
                                                      array, 730, 732
allocator, 508
                                                          begin, 730
    address, 508, 509
                                                          data, 732
    allocate, 509
                                                          end, 730
    constructor, 509
                                                          fill, 732
    deallocate, 509
                                                          get, 732, 733
    destructor, 509
                                                          max_size, 730
    max_size, 509
                                                          size, 730, 732
    operator!=, 509
```

swap, 731, 732	exchange, 1108
asin, 944, 956	fetch_, 1110
complex, 884	load, 1108
asinh, 956	operator @=, 1110
complex, 884	operator $C$ , 1108
<pre><assert.h>, 409</assert.h></pre>	operator++, 1110
assign	operator, 1110, 1111
deque, 736	operator=, 1107
list, 748	store, 1107
basic_regex, 1068	atomic_compare_exchange_strong
basic_string, 623, 624	atomic type, 1108
deque, 736	shared_ptr, 538
error_code, 466	atomic_compare_exchange_strong_explicit
error_condition, 469	atomic type, 1108
forward_list, 741	shared_ptr, 538
function, 556	atomic_compare_exchange_weak
list, 748	atomic type, 1108
vector, 755	shared_ptr, 538
async, 1155	atomic_compare_exchange_weak_explicit
at	atomic type, 1108
basic_string, 621	shared_ptr, 538
map, 766	atomic_exchange
unordered_map, 782	atomic type, 1108
at_quick_exit, 438, 439	shared_ptr, 538
atan, 944, 956	atomic_exchange_explicit
complex, 884	atomic type, 1108
atan2, 944, 956	shared_ptr, 538
atanh, 956	atomic_fetch_
complex, 884	atomic type, 1110
atexit, 61, 408, 438	atomic_flag
<atomic>, 1096</atomic>	clear, 1112
atomic type	atomic_flag_clear, 1112
atomic_compare_exchange_strong, 1108	atomic_flag_clear_explicit, 1112
atomic_compare_exchange_strong	atomic_flag_test_and_set, 1111
explicit, 1108	atomic_flag_test_and_set_explicit, 1111
atomic_compare_exchange_weak, 1108	atomic_is_lock_free
atomic_compare_exchange_weak	atomic type, 1107
explicit, 1108	shared_ptr, 537
atomic_exchange, 1108	atomic_load
atomic_exchange_explicit, 1108	atomic type, 1108
atomic_fetch_, 1110	shared_ptr, 537
atomic_is_lock_free, 1107	atomic_load_explicit
atomic_load, 1108	atomic type, 1108
atomic_load_explicit, 1108	shared_ptr, 537
atomic_store, 1107	atomic_signal_fence, 1112
atomic_store_explicit, 1107	atomic_store
compare_exchange_strong, 1108	atomic type, 1107
compare_exchange_strong_explicit, 1108	shared_ptr, 537
compare_exchange_weak, 1108	atomic_store_explicit
compare_exchange_weak_explicit, 1108	atomic type, 1107
constructor, 1106, 1107	shared_ptr, 537
•	_ <b>_</b>

atomic_thread_fence, 1112	constructor, 1034
$auto_ptr, 528, 1218$	destructor, $1035$
$\mathtt{auto\_ptr},1219$	operator=, $1035$
auto_ptr_ref, 1220	$\mathtt{swap},1035$
constructor, 1219, 1220	$basic_filebuf < char >, 1032$
destructor, 1220	$basic_filebuf, 1032$
operator=, $1219$	$\texttt{basic\_fstream},961,1044$
auto_ptr_ref	${\tt basic\_fstream},1045$
auto_ptr, 1220	constructor, 1045
operator auto_ptr, 1220	operator=, $1045$
operator=, 1220	$\overline{\mathtt{swap}}, 1045$
-	basic_ifstream, 961, 1039
b	basic_ifstream, 1040
cauchy_distribution, 923	constructor, 1040
extreme_value_distribution, 920	operator=, 1041
uniform_int_distribution, 911	swap, $1041$
uniform_real_distribution, 912	basic_ifstream <char>, 1032</char>
weibull_distribution, 919	basic_ifstream <wchar_t>, 1032</wchar_t>
back	basic_ios, 961, 976
basic_string, 621	basic_ios, 977
back_insert_iterator, 820	constructor, 977
back_insert_iterator, 820	destructor, 977
back_inserter, 821	exceptions, 981
bad	<u>-</u>
basic_ios, 980	fill, 978
bad_alloc, 108, 440, 443, 444	init, 977
bad_alloc, 444	move, 979
bad_alloc::what	rdbuf, 978
implementation-defined, 444	set_rdbuf, 979
bad_array_new_length, 444	swap, 979
bad_array_new_length, 444	tie, 978
bad_cast, 98, 445, 446	basic_ios <char>, 966</char>
	basic_ios <wchar_t>, 966</wchar_t>
bad_cast, 446	basic_iostream, 1007
bad_cast::what implementation-defined, 446	basic_iostream, 1008
,	constructor, 1008
bad_exception, 449	destructor, 1008
bad_exception, 449	operator=, 1008
bad_exception::what	$\mathtt{swap},1008$
implementation-defined, 449	$\texttt{basic\_istream},961,995$
bad_function_call, 553	$ exttt{basic_istream},997$
bad_function_call, 553	constructor, 997
bad_typeid, 99, 445, 447	destructor, 997, 999
bad_typeid, 447	$\mathtt{get},1002,1003,1007$
bad_weak_ptr, 523	operator $<<$ , $1008$
bad_weak_ptr, 524	operator=, $998$
$\mathtt{what}, 524$	$\mathtt{seekg},1006$
base	$\mathtt{swap},998$
move_iterator, 825	$\mathtt{tellg}, 1006$
reverse_iterator, 817	$basic_istream < char >, 994$
$\texttt{basic\_filebuf}, 961, 1033$	$basic_istream < wchar_t >, 994$
${\tt basic\_filebuf}, 1034$	basic_istreambuf_iterator, 961

basic_istringstream, 961, 1026	gyan 080
basic_istringstream, 1027	<pre>swap, 989 basic_streambuf<char>, 984</char></pre>
constructor, 1027	basic_streambuf <wchar_t>, 984</wchar_t>
operator=, 1028	basic_string, 610, 611, 633, 1021
str, 1028	append, 622, 623
swap, 1028	assign, 623, 624
basic_istringstream <char>, 1021</char>	at, 621
basic_istringstream <wchar_t>, 1021</wchar_t>	back, 621
basic_ofstream, 961, 1042	begin, 619
basic_ofstream, 1042	capacity, 620
constructor, 1042, 1043	cbegin, 619
operator=, 1043	cend, 619
swap, 1043	clear, 620
basic_ofstream <char>, 1032</char>	compare, 633
basic_ofstream <wchar_t>, 1032</wchar_t>	constructor, 615–617
basic_ostream, 961, 1076	copy, 628
basic_ostream, 1010	crbegin, 619
constructor, 1011	crend, 619
destructor, 1010, 1011	empty, 620
operator<<, 1014, 1015, 1017	end, 619
operator=, 1011	erase, 626
seekp, 1012	find, 629
swap, 1011	find_first_not_of, 631, 632
basic_ostream <char>, 995</char>	find_first_of, 630
basic_ostream <wchar_t>, 995</wchar_t>	find_last_not_of, 632
basic_ostreambuf_iterator, 961	find_last_of, 631
basic_ostringstream, 961, 1028	front, 621
basic_ostringstream, 1029	get_allocator, 629
constructor, 1029	getline, 638, 639
operator=, 1029	insert, 624, 625
str, 1030	length, 619
swap, 1030	$\mathtt{max\_size}, 620$
basic_ostringstream <char>, 1021</char>	operator!=, 636
basic_ostringstream <wchar_t>, 1021</wchar_t>	operator+, 633-635
basic_regex, 1051, 1064, 1093	operator+=, 621, 622
assign, 1068	operator<, 636
basic_regex, 1066, 1067	operator<=, 637
constants, 1065, 1066	operator<<, 638
constructor, 1066, 1067	operator=, 618, 619
$flag_type, 1069$	operator==, 635
$\mathtt{getloc},1069$	operator>, $636$ , $637$
imbue, $1069$	operator>=, $637$
mark_count, 1069	operator>>, $638$
operator=, $1067$	$ exttt{operator[]},621$
$\mathtt{swap},1069$	pop_back, $626$
${\tt basic\_streambuf},961,985$	${\tt push\_back},623$
${\tt basic\_streambuf},987$	${ t rbegin}, 619$
constructor, 987	$\mathtt{rend},619$
destructor, 987	$\mathtt{replace},626628$
operator=, $989$	$\mathtt{reserve},620$
$\mathtt{setbuf},1026$	$\mathtt{resize}, 620$

rfind, 630	$\mathtt{bit\_or},549$
$\mathtt{shrink\_to\_fit},620$	$\mathtt{bit\_xor},549$
$\mathtt{size},619$	  bitset>, $490$
$\mathtt{substr},632$	$\mathtt{bitset}, 490$
$\mathtt{swap},629,637$	$\mathtt{bitset},492,493$
$\verb basic_stringbuf , 961, 1021 $	$\mathtt{flip},495$
$ exttt{basic\_stringbuf}, 1023$	operator[], $496$
constructor, $1023$	$\mathtt{reset}, 494$
operator=, $1023$	$\mathtt{set}, 494$
str,1024	boolalpha, $981$
$\mathtt{swap},1024$	byte_string
basic_stringbuf <char>, 1021</char>	wstring_convert, 655
basic_stringbuf <wchar_t>, 1021</wchar_t>	
basic_stringstream, 961, 1030	c_str
${\tt basic\_stringstream},1031$	$\mathtt{basic\_string}, 629$
constructor, 1031	cacos
operator=, $1031$	$\mathtt{complex},884$
str,1032	cacosh
swap, 1031, 1032	$\mathtt{complex},884$
before	$\mathtt{call\_once},1134$
type_info, 446	calloc, 512, 1201
before_begin	capacity
forward_list, 741	$\mathtt{basic\_string}, 620$
begin, $452$	vector, 756
array, 730	casin
basic_string, 619	complex, 884
initializer_list, $453$	casinh
match_results, 1080	complex, $884$
valarray, 952	<cassert>, 409</cassert>
begin(C&), 835	catan
begin(initializer_list <e>), 453</e>	complex, 884
begin(T (&)[N]), 835	catanh
bernoulli_distribution, 912	complex, 884
constructor, 912	category
p, 913	error_code, 467
beta	error_condition, 469
gamma_distribution, 918	locale, 648
bidirectional_iterator_tag, 813	cauchy_distribution, 922
binary_function, 545, 1214	a, 923
binary_negate, 550	b, 923
binary_search, 863	constructor, 923
bind, 550-552	cbefore_begin
	forward_list, 741
bind1st, 1217 bind2nd, 1218	cbegin
•	basic_string, 619
binder1st, 1217	cbrt, 956
binder2nd, 1217	<pre><ccomplex>, 886</ccomplex></pre>
binomial_distribution, 913	cend
constructor, 913	
p, 913	basic_string, 619
t, 913	cerr, 964 <cerrno>, 419</cerrno>
$\mathtt{bit\_and},\ 549$	(CG11HO), 413

<cfenv>, 875</cfenv>	do_out, 667
CHAR_BIT, 436	do_unshift, 668
char_class_type	encoding, 667
regex_traits, 1062	in, 667
CHAR_MAX, 436	length, $667$
$\mathtt{char\_traits},603606$	$\mathtt{max\_length}, 667$
$\mathtt{char\_type},\ 603$	out,666
$\mathtt{int\_type},603$	$\mathtt{unshift}, 667$
${\tt off\_type},603$	${\tt codecvt\_byname},669$
${ t pos\_type,603}$	collate, 681
${ t state\_type}, 603$	$\mathtt{compare}, 681$
char_type	$ exttt{do\_compare},682$
$\mathtt{char\_traits},603$	$do_hash, 682$
${ t chi\_squared\_distribution}, 922$	$ ext{do\_transform},682$
constructor, 922	$\mathtt{hash}, 682$
n, 922	${ t transform},682$
chrono, $578$	collate_byname, 682
cin, 964	combine
<ciso646>, 1199</ciso646>	locale, $652$
classic	common_type, 582, 586
locale, $653$	compare
classic_table	basic_string, 633
ctype <char>, 665</char>	collate, 681
clear	sub_match, 1070
atomic_flag, 1112	compare_exchange_strong
basic_ios, 980	atomic type, 1108
basic_string, 620	compare_exchange_strong_explicit
error_code, 467	atomic type, 1108
error_condition, 469	compare_exchange_weak
forward_list, 743	atomic type, 1108
<climits>, 1206</climits>	compare_exchange_weak_explicit
<clocale>, 406, 1199</clocale>	atomic type, 1108
clock, 453, 454	<pre><complex>, 876</complex></pre>
clock_t, 454	complex, 878
CLOCKS_PER_SEC, 454	complex, 880
clog, 964	imag, 880
close	operator-, 882
basic_filebuf, 1036, 1046	operator/, 882
basic_ifstream, 1041	real, 880
basic_ofstream, 1044	<pre><condition_variable>, 1136</condition_variable></pre>
messages, 695	condition_variable
code	constructor, 1137
future_error, 1146	destructor, 1137
system_error, 471	notify_all, 1138
codecvt, 665, 699	notify_one, 1137
always_noconv, 667	wait, 1138
do_always_noconv, 669	wait_for, 1139, 1140
do_encoding, 668	wait_101, 1139, 1140 wait_until, 1138, 1139
do_in, 667	condition_variable_any
do_ln, 669	constructor, 1141
do_max_length, 669	destructor, 1141
do_max_rengen, 003	destructor, 1141

notify all 1149	Coatdints 126
notify_all, 1142	<pre><cstdint>, 436 <cstdio>, 963-965, 1033, 1035, 1036, 1199</cstdio></cstdint></pre>
notify_one, 1142	<pre><cstd1b>, 408, 453, 454, 1199, 1202</cstd1b></pre>
wait, 1142 wait_for, 1143	<pre><cstuffb>, 406, 453, 454, 1199, 1202</cstuffb></pre> <cstring>, 406, 1199, 1206, 1211</cstring>
wait_not, 1143 wait_until, 1142, 1143	<pre><cstiling>, 400, 1199, 1200, 1211 <ctgmath>, 955</ctgmath></cstiling></pre>
	<pre><ctime>, 453, 454, 646, 1199</ctime></pre>
conj, 886	ctype, 659
complex, 884	
const_mem_fun1_ref_t, 1216	do_is, 661
const_mem_fun1_t, 1216 const_mem_fun_ref_t, 1216	do_narrow, 662 do_scan_not, 661
const_mem_fun_t, 1216	do_tolower, 661
const_pointer	do_tolower, 661
allocator_traits, 506	do_toupper, oor do_widen, 661
const_pointer_cast	is, 660
<del></del>	
shared_ptr, 532	narrow, 660
const_void_pointer	scan_is, 660
allocator_traits, 506	scan_not, 660 tolower, 660
construct	
scoped_allocator_adaptor, 597, 598 converted	toupper, $660$ widen, $660$
wstring_convert, 655	ctype <char>, 662 classic_table, 665</char>
copy, 851	<del>-</del> ,
basic_string, 628	constructor, 664
copy_backward, 851	ctype <char>, 664</char>
copy_n, 851	destructor, 663 do_narrow, 665
copyfmt basic_ios, 978	do_tolower, 665
copysign, 956	do_toupper, 665
cos, 944, 956	do_widen, 665
complex, 885	is, 664
cosh, 944, 956	narrow, 665
complex, 885	scan_is, 664
count, 848	scan_not, 664
bitset, 495	table, 665
duration, 584	tolower, 664
count_if, 848	toupper, 664
cout, 964	widen, 664
crbegin	ctype_base, 658
basic_string, 619	do_scan_is, 661
cref	ctype_byname, 662
reference_wrapper, 546	<pre><cuchar>, 419</cuchar></pre>
crend	curr_symbol
basic_string, 619	moneypunct, 693
<pre><csetjmp>, 419, 453, 454</csetjmp></pre>	current_exception, 450
cshift	<pre><cwchar>, 419, 1199</cwchar></pre>
valarray, 941	,,
<pre><csignal>, 453, 454</csignal></pre>	data
<pre><cstdalign>, 454</cstdalign></pre>	$ exttt{basic\_string}, 629$
<pre><cstdarg>, 419, 453, 454</cstdarg></pre>	array, 732
<pre><cstdbool>, 453, 454</cstdbool></pre>	vector, 757
<pre><cstddef>, 1199, 1201</cstddef></pre>	date_order
, ,	

+: COA	
time_get, 684	pointer_traits, 502
DBL_DIG, 436	digits
DBL_EPSILON, 436	numeric_limits, 429
DBL_MANT_DIG, 436	digits10
DBL_MAX, 436	numeric_limits, 429
DBL_MAX_10_EXP, 436	discard_block_engine, 902
DBL_MAX_EXP, 436	constructor, 903
DBL_MIN, 436	discrete_distribution, 925
DBL_MIN_10_EXP, 436	constructor, 926
DBL_MIN_EXP, 436	probabilities, 926
deallocate	distance, 814
allocator, 509	div, 956
allocator_traits, 507	divides, 546
scoped_allocator_adaptor, 596	do_always_noconv
dec, 983, 1014	codecvt, 669
DECIMAL_DIG, 436	do_close
decimal_point	$\mathtt{message}, 696$
moneypunct, 693	do_compare
numpunct, 679	collate, 682
${\tt declare\_no\_pointers},503$	do_curr_symbol
$declare\_reachable, 503$	moneypunct, $694$
declval, 476	do_date_order
default_delete	$\mathtt{time\_get}, 685$
${\tt default\_delete},  514$	do_decimal_point
$\mathtt{operator()},514,515$	$\mathtt{moneypunct}, 693$
default_error_condition	$\mathtt{numpunct},680$
$error\_category, 464$	do_encoding
${ t error\_code},467$	$\mathtt{codecvt}, 668$
default_random_engine, 906	do_falsename
defaultfloat, 983	$\mathtt{numpunct},680$
delete	do_frac_digits
operator, $512$	moneypunct, 694
operator, $420, 441, 442$	do_get
denorm_absent, 434	${\tt messages},696$
denorm_indeterminate, 434	$\mathtt{money\_get},689$
denorm_min	$\mathtt{num\_get},671,674$
numeric_limits, 432	$\mathtt{time\_get},686$
denorm_present, 434	do_get_date
densities	${\tt time\_get},685$
piecewise_constant_distribution, 928	do_get_monthname
$piecewise\_linear\_distribution, 930$	${\tt time\_get},685$
<deque>, 728</deque>	do_get_time
deque, 733	$time\_get, 685$
assign, $736$	do_get_weekday
deque, 735	$\texttt{time\_get},685$
shrink_to_fit, 736	do_get_year
swap, 737	time_get, 686
detach	do_grouping
thread, 1121	moneypunct, 694
difference_type	numpunct, 680
allocator_traits, 506	do_hash
_ ,	_

do_in	collate, 682	domain_error, 457
codecvt, 667		
do_is	_	
ctype, 661		
do_length		
codecvt, 669		·
do_max_length         operator*, 586           codect, 669         operator*, 585           do_narrow, 665         operator*, 584, 590           ctype, 662         operator*, 584           ctype, 662         operator*, 584           do_neg_format         operator*, 585           moneypunct, 694         operator*, 585           do_open         operator*, 586, 587           do_open         operator*, 587           do_out         operator*, 587           codect, 667         operator*, 587           do_pos_format         operator*, 587           moneypunct, 694         operator*, 587           do_pos_format         operator*, 587           moneypunct, 694         operator*, 587           do_pos_format         operator*, 587           moneypunct, 694         operator*, 588           do_postive_sign         operator*, 588           moneypunct, 694         duration_cast, 588           do_put         duration_cast, 588           time_put, 675, 678         max, 582           time_put, 675, 678         max, 582           time_put, 688         duration_cast, 581           do_scan_not         share_put, 681           ctype, 661         do_touser <tr< td=""><td></td><td>·</td></tr<>		·
codecvt, 669 do_narrow, 665 ctype, 662 ctype, 662 ctype, 662 ctype, 665 ctype, 665 ctype, 665 ctype, 665 do_neg_format moneypunct, 694 do_negative_sign moneypunct, 694 do_perator+, 585 do_perator, 585 do_negative_sign moneypunct, 694 do_open messages, 695 do_out codecvt, 667 do_pos_format moneypunct, 694 do_put moneypunct, 694 do_put moneypunct, 696 do_scan_is ctype_base, 661 do_scan_not ctype, 661 do_thousands_sep moneypunct, 680 do_tolower ctype, 661 ctype <char>, 665 do_truename collate, 682 do_widen, 665 ctype, 661 constructor, 536 constructor,</char>	•	
do_narrow, 665     ctype, 662     ctype, 662     ctype <char>, 665     ctype<char>, 665     do_neg_format     moneypunct, 694     do_negative_sign     moneypunct, 694     do_open     messages, 695     do_nos_format     moneypunct, 687     do_pos_format     moneypunct, 688     do_pos_format     moneypunct, 694     do_open     messages, 695     do_out     codecvt, 667     do_pos_format     moneypunct, 694     do_positive_sign     moneypunct, 695     do_positive_sign     moneypunct, 696     do_positive_sign     moneypunct, 698     do_positive_sign     moneypunct, 6991     num_put, 675, 678     time_put, 688     do_scan_is     ctype_base, 661     do_tousands_sep     moneypunct, 693     numpunct, 680     do_tolower     ctype, 661     ctype<char>, 665     do_toupper     ctype, 661     ctype<char>, 665     do_toupper     ctype<char>, 665     do_truename     collate, 682     do_truename     rumpunct, 680     do_unshift     codecvt, 668     do_widen, 665     ctype, 661     cviden, 665     ctype, 661     constructor, 536     constructor, 536</char></char></char></char></char>		_
ctype, 662         operator++, 584           ctype <char>, 665         operator+=, 585           do_neg_format         operator-, 584, 590           moneypunct, 694         operator, 585           do_negative_sign         operator/-, 585           moneypunct, 694         operator/-, 586, 587           do_open         operator/-, 586, 587           do_out         operator&lt;-, 587</char>	•	
ctype <char>, 665         operator+=, 585           do_neg_format         operator-, 584, 590           moneypunct, 694         operator-=, 585           do_negative_sign         operator/, 586, 587           do_open         operator/, 586           messages, 695         operator&lt;, 587</char>		
do_neg_format         operator-, 584, 590           moneypunct, 694         operator, 585           do_negative_sign         operator-, 585           moneypunct, 694         operator/, 586, 587           do_open         operator-, 585           messages, 695         operator-, 587           do_out         operator-, 587           codecvt, 667         operator-, 587           do_pos_format         operator-, 588           moneypunct, 694         operator-, 585           do_positive_sign         operator-, 585           moneypunct, 694         operator-, 585           do_positive_sign         operator-, 587           moneypunct, 694         operator-, 587           do_positive_sign         operator-, 588           moneypunct, 694         operator-, 587           do_positive_sign         operator-, 587           moneypunct, 694         operator-, 588           do_positive_sign         operator-, 585           do_solouser         duration_cast           do_toune_non		·
moneypunct, 694 do_negative_sign moneypunct, 694 do_open moneypunct, 694 do_open messages, 695 do_out codecvt, 667 do_pos_format moneypunct, 694 do_pos_format moneypunct, 694 do_positive_sign moneypunct, 694 do_put money_put, 691 num_put, 675, 678 time_put, 688 do_scan_is ctype_base, 661 do_thousands_sep moneypunct, 693 numpunct, 680 do_tolower ctype, 661 ctype <char>, 665 do_transform collate, 682 do_unshift codecvt, 668 do_widen, 665 ctype, 661 do_widen, 665 ctype, 661 do_widen, 665 ctype, 661 do_massages moneypunct, 680 do_unshift codecvt, 668 do_widen, 665 ctype, 661 constructor, 536 do_punct, 665 do_tounstructor, 536 do_massic_string, 620 match_results, 1079 enable_shared_from_this, 536 constructor, 536</char>		· ,
do_negative_sign         operator-, 585           moneypunct, 694         operator/, 586, 587           do_open         operator/=, 585           messages, 695         operator<-, 587		_
moneypunct, 694 do_open messages, 695 do_out codecvt, 667 do_pos_format moneypunct, 694 do_pos_format moneypunct, 694 do_positive_sign moneypunct, 694 do_put moneypunct, 694 do_put moneypunct, 691 do_put moneypunct, 691 do_put moneypunct, 691 mum_put, 675, 678 time_put, 688 do_scan_is ctype_base, 661 do_thousands_sep moneypunct, 693 numpunct, 693 numpunct, 680 do_tolower ctype, 661 ctype <char> ctype, 661 ctype<char> ctype, 661 ctype<char> ctype, 661 ctype<char> ctype, 662 do_trunsform collate, 682 do_unshift codecvt, 668 do_widen, 665 ctype, 661 ctype, 661 constructor, 536 con</char></char></char></char>		- · · · · · · · · · · · · · · · · · · ·
do_open         operator/=, 585           messages, 695         operator<, 587		
messages, 695 do_out	-	
do_out	<del></del>	- <u>-</u>
codecvt, 667	messages, 695	·
do_pos_format         operator>=, 588           moneypunct, 694         operator%, 587           do_positive_sign         operator%=, 585           moneypunct, 694         zero, 586           do_put         duration_cast, 588           money_put, 691         duration_values, 581           num_put, 675, 678         max, 582           time_put, 688         min, 582           do_scan_is         zero, 581           ctype_base, 661         dynamic_pointer_cast           do_scan_not         shared_ptr, 532           ctype, 661         eback           do_tousands_sep         basic_streambuf, 990           moneypunct, 693         egptr           do_tolower         ctype, 661           ctype, 661         element_type           ctype, 661         pointer_traits, 502           do_toupper         emplace           ctype, 661         priority_queue, 799           emplace_after         forward_list, 742           do_transform         forward_list, 742           collate, 682         emplace_front           do_truename         forward_list, 741           numpunct, 680         empty, 813           do_unshift         basic_string, 620           cod	_	
moneypunct, 694 do_positive_sign	$\mathtt{codecvt},667$	operator==, $587$
do_positive_sign         operator%=, 585           moneypunct, 694         zero, 586           do_put         duration_cast, 588           money_put, 691         max, 582           num_put, 675, 678         max, 582           time_put, 688         min, 582           do_scan_is         zero, 581           ctype_base, 661         dynamic_pointer_cast           do_scan_not         shared_ptr, 532           ctype, 661         basic_streambuf, 990           do_thousands_sep         basic_streambuf, 990           moneypunct, 680         egptr           do_tolower         basic_streambuf, 990           ctype, 661         element_type           ctype         pointer_traits, 502           do_tolower         emplace           ctype         emplace           ctype         enplace           ctype         fof1           ctype         priority_queue, 799           emplace_after         forward_list, 742           collate, 682         emplace_front           do_truename         forward_list, 741           numpunct, 680         empty, 813           basic_string, 620         match_results, 1079           do_widen, 665         enable_share	do_pos_format	· ,
moneypunct, 694  do_put	${\tt moneypunct}, 694$	operator $\%,587$
do_put       duration_cast, 588         money_put, 691       duration_values, 581         num_put, 675, 678       max, 582         time_put, 688       min, 582         do_scan_is       zero, 581         ctype_base, 661       dynamic_pointer_cast         do_scan_not       shared_ptr, 532         ctype, 661       eback         do_thousands_sep       basic_streambuf, 990         moneypunct, 693       egptr         do_tolower       basic_streambuf, 990         ctype, 661       element_type         ctype, 661       pointer_traits, 502         do_toupper       emplace         ctype, 661       priority_queue, 799         ctype       emplace_after         forward_list, 742       emplace_front         do_truename       forward_list, 741         numpunct, 680       empty, 813         do_ushift       basic_string, 620         codecvt, 668       match_results, 1079         do_widen, 665       enable_shared_from_this, 536         ctype, 661       constructor, 536	do_positive_sign	operator $\%$ =, $585$
money_put, 691 num_put, 675, 678 time_put, 688 do_scan_is ctype_base, 661 do_scan_not ctype, 661 do_thousands_sep moneypunct, 693 numpunct, 680 do_tolower ctype, 661 ctype <char>, 665 do_toupper ctype, 661 ctype<char>, 665 do_transform collate, 682 do_truename numpunct, 680 do_unden, 665 ctype, 661 constructor, 536 do_widen, 665 ctype, 661 constructor, 536 duration_values, 581 max, 582 min, 582 demax, 582 dimax, 582</char></char>	$\mathtt{moneypunct}, 694$	${\tt zero}, 586$
num_put, 675, 678         max, 582           time_put, 688         min, 582           do_scan_is         zero, 581           ctype_base, 661         dynamic_pointer_cast           do_scan_not         shared_ptr, 532           ctype, 661         do_thousands_sep           moneypunct, 693         basic_streambuf, 990           numpunct, 680         egptr           do_tolower         basic_streambuf, 990           ctype, 661         element_type           ctype, 661         pointer_traits, 502           do_toupper         emplace           ctype, 661         priority_queue, 799           ctype         emplace_after           forward_list, 742         emplace_front           do_truename         forward_list, 741           numpunct, 680         empty, 813           do_unshift         basic_string, 620           codecvt, 668         match_results, 1079           do_widen, 665         enable_shared_from_this, 536           ctype, 661         constructor, 536	do_put	$\mathtt{duration\_cast},588$
time_put, 688  do_scan_is	money_put, 691	${ t duration\_values,581}$
do_scan_is       zero, 581         ctype_base, 661       dynamic_pointer_cast         do_scan_not       shared_ptr, 532         ctype, 661       eback         do_thousands_sep       basic_streambuf, 990         numpunct, 680       egptr         do_tolower       basic_streambuf, 990         ctype, 661       element_type         ctype       pointer_traits, 502         do_toupper       emplace         ctype       priority_queue, 799         ctype       emplace_after         forward_list, 742       emplace_front         do_truename       forward_list, 741         numpunct, 680       empty, 813         do_unshift       basic_string, 620         codecvt, 668       match_results, 1079         do_widen, 665       enable_shared_from_this, 536         ctype, 661       constructor, 536	$\mathtt{num\_put},675,678$	$\mathtt{max},582$
ctype_base, 661       dynamic_pointer_cast         do_scan_not       shared_ptr, 532         ctype, 661       eback         do_thousands_sep       basic_streambuf, 990         moneypunct, 680       egptr         do_tolower       basic_streambuf, 990         ctype, 661       element_type         ctype <char>, 665       pointer_traits, 502         do_toupper       emplace         ctype<char>, 665       emplace_after         do_transform       forward_list, 742         collate, 682       emplace_front         do_truename       forward_list, 741         numpunct, 680       empty, 813         do_unshift       basic_string, 620         codecvt, 668       match_results, 1079         do_widen, 665       enable_shared_from_this, 536         ctype, 661       constructor, 536</char></char>	$time_put, 688$	$\mathtt{min},582$
do_scan_not       shared_ptr, 532         ctype, 661       eback         moneypunct, 693       basic_streambuf, 990         numpunct, 680       egptr         do_tolower       basic_streambuf, 990         ctype, 661       element_type         ctype       pointer_traits, 502         do_toupper       emplace         ctype, 661       priority_queue, 799         ctype       emplace_after         do_transform       forward_list, 742         collate, 682       emplace_front         do_truename       forward_list, 741         numpunct, 680       empty, 813         do_unshift       basic_string, 620         codecvt, 668       match_results, 1079         do_widen, 665       enable_shared_from_this, 536         ctype, 661       constructor, 536	do_scan_is	$\mathtt{zero},581$
ctype, 661       do_thousands_sep moneypunct, 693 moneypunct, 693 mumpunct, 680       basic_streambuf, 990 egptr basic_streambuf, 990 element_type ctype, 661 element_type pointer_traits, 502 emplace priority_queue, 799 emplace after forward_list, 742 emplace_after forward_list, 742 emplace_front forward_list, 741 empty, 813 do_unshift codecvt, 668 do_widen, 665 ctype, 661       empty, 813 basic_string, 620 match_results, 1079 enable_shared_from_this, 536 constructor, 536	ctype_base, 661	dynamic_pointer_cast
do_thousands_sep       eback         moneypunct, 693       basic_streambuf, 990         numpunct, 680       egptr         do_tolower       basic_streambuf, 990         ctype, 661       element_type         ctype       pointer_traits, 502         do_toupper       emplace         ctype       priority_queue, 799         ctype       emplace_after         do_transform       forward_list, 742         collate, 682       emplace_front         do_truename       forward_list, 741         numpunct, 680       empty, 813         do_unshift       basic_string, 620         codecvt, 668       match_results, 1079         do_widen, 665       enable_shared_from_this, 536         ctype, 661       constructor, 536	do_scan_not	${ t shared\_ptr}, 532$
moneypunct, 693 numpunct, 680  do_tolower ctype, 661 ctype <pre>ctype</pre> do_toupper ctype, 661 ctype ctype, 661 ctype do_toupper ctype emplace ctype, 661 ctype do_transform collate, 682 do_truename numpunct, 680  do_unshift codecvt, 668 do_widen, 665 ctype, 661  basic_streambuf, 990 emptr basic_streambuf, 990 element_type pointer_traits, 502 emplace priority_queue, 799 emplace_after forward_list, 742 emplace_front forward_list, 741 empty, 813 basic_string, 620 match_results, 1079 enable_shared_from_this, 536 constructor, 536	ctype, 661	
numpunct, 680  do_tolower ctype, 661 ctype ctype do_toupper ctype, 661 ctype do_toupper ctype, 661 ctype char>, 665  do_toupper ctype, 661 ctype char>, 665  do_transform collate, 682  do_truename numpunct, 680  do_unshift codecvt, 668  do_widen, 665 ctype, 661  egptr basic_streambuf, 990 element_type pointer_traits, 502 emplace priority_queue, 799 emplace_after forward_list, 742 emplace_front forward_list, 741 empty, 813 basic_string, 620 match_results, 1079 enable_shared_from_this, 536 constructor, 536	do_thousands_sep	eback
do_tolower basic_streambuf, 990 ctype, 661 element_type ctype <char>, 665 pointer_traits, 502 do_toupper emplace ctype, 661 priority_queue, 799 ctype<char>, 665 emplace_after do_transform forward_list, 742 collate, 682 emplace_front do_truename forward_list, 741 numpunct, 680 empty, 813 do_unshift codecvt, 668 do_widen, 665 ctype, 661 enable_shared_from_this, 536 cconstructor, 536</char></char>	moneypunct, $693$	$\mathtt{basic\_streambuf},990$
do_tolower     ctype, 661     ctype ctype ctype ctype ctype ctype, 665  do_toupper     ctype, 661     ctype char>, 665  do_talpe ctype ctype char>, 665  do_transform     collate, 682  do_truename     numpunct, 680  do_unshift     codecvt, 668  do_widen, 665     ctype, 661  do_widen, 665     ctype, 661  do_struename     constructor, 536  do_struename     constructor, 536  constructor, 536	numpunct, 680	egptr
ctype <char>, 665  do_toupper ctype, 661 ctype<char>, 665  do_transform collate, 682  do_truename numpunct, 680  do_unshift codecvt, 668  do_widen, 665 ctype, 661  priority_queue, 799 emplace_after forward_list, 742 emplace_front forward_list, 741 empty, 813 basic_string, 620 match_results, 1079 enable_shared_from_this, 536 constructor, 536</char></char>		$\mathtt{basic\_streambuf},990$
do_toupper       emplace         ctype, 661       priority_queue, 799         ctype <char>, 665       emplace_after         do_transform       forward_list, 742         collate, 682       emplace_front         do_truename       forward_list, 741         numpunct, 680       empty, 813         do_unshift       basic_string, 620         codecvt, 668       match_results, 1079         do_widen, 665       enable_shared_from_this, 536         ctype, 661       constructor, 536</char>	ctype, 661	
do_toupper       emplace         ctype, 661       priority_queue, 799         ctype <char>, 665       emplace_after         do_transform       forward_list, 742         collate, 682       emplace_front         do_truename       forward_list, 741         numpunct, 680       empty, 813         do_unshift       basic_string, 620         codecvt, 668       match_results, 1079         do_widen, 665       enable_shared_from_this, 536         ctype, 661       constructor, 536</char>	ctype <char>, 665</char>	pointer_traits, $502$
ctype, 661     priority_queue, 799       ctype <char>, 665     emplace_after       do_transform     forward_list, 742       collate, 682     emplace_front       do_truename     forward_list, 741       numpunct, 680     empty, 813       do_unshift     basic_string, 620       codecvt, 668     match_results, 1079       do_widen, 665     enable_shared_from_this, 536       ctype, 661     constructor, 536</char>		emplace
ctype <char>, 665       emplace_after         do_transform       forward_list, 742         collate, 682       emplace_front         do_truename       forward_list, 741         numpunct, 680       empty, 813         do_unshift       basic_string, 620         codecvt, 668       match_results, 1079         do_widen, 665       enable_shared_from_this, 536         ctype, 661       constructor, 536</char>	<del></del>	$priority_queue, 799$
do_transform       forward_list, 742         collate, 682       emplace_front         do_truename       forward_list, 741         numpunct, 680       empty, 813         do_unshift       basic_string, 620         codecvt, 668       match_results, 1079         do_widen, 665       enable_shared_from_this, 536         ctype, 661       constructor, 536		emplace_after
$\begin{array}{cccc} \text{collate, } 682 & \text{emplace\_front} \\ \text{do\_truename} & \text{forward\_list, } 741 \\ \text{numpunct, } 680 & \text{empty, } 813 \\ \text{do\_unshift} & \text{basic\_string, } 620 \\ \text{codecvt, } 668 & \text{match\_results, } 1079 \\ \text{do\_widen, } 665 & \text{enable\_shared\_from\_this, } 536 \\ \text{ctype, } 661 & \text{constructor, } 536 \end{array}$	* <del>-</del>	${ t forward\_list}, 742$
$\begin{array}{llllllllllllllllllllllllllllllllllll$	_	emplace_front
numpunct, 680       empty, 813         do_unshift       basic_string, 620         codecvt, 668       match_results, 1079         do_widen, 665       enable_shared_from_this, 536         ctype, 661       constructor, 536		$forward\_list, 741$
$\begin{array}{lll} \texttt{do\_unshift} & \texttt{basic\_string}, 620 \\ \texttt{codecvt}, 668 & \texttt{match\_results}, 1079 \\ \texttt{do\_widen}, 665 & \texttt{enable\_shared\_from\_this}, 536 \\ \texttt{ctype}, 661 & \texttt{constructor}, 536 \end{array}$	<del>-</del>	empty, <u>813</u>
codecvt, 668 match_results, 1079 do_widen, 665 enable_shared_from_this, 536 ctype, 661 constructor, 536	<b>-</b> ,	basic_string, 620
do_widen, 665 enable_shared_from_this, 536 ctype, 661 constructor, 536	_	
ctype, 661 constructor, 536		
Jr-,		
ctype <char>, 665 destructor, 536</char>		destructor, 536
domain_error, 456, 457 operator=, 536	* <del>-</del>	
40main_61161, 400, 401	domain_61101, 400, 401	1 , 555

shared_from_this, 536 encoding    codecvt, 667 end, 452    array, 730    basic_string, 619    initializer_list, 453    match_results, 1080    valarray, 952 end(C&), 835	assign, 466 category, 467 clear, 467 default_error_condition, 467 error_code, 466 message, 467 operator bool, 467 operator!=, 470 operator<, 467 operator<<, 467
end(initializer_list <e>), 453</e>	operator=, $466$
end(T (&)[N]), 835	operator==, $470$
endl, 1014, 1016	value, 467
ends, $1016$	${ t error\_condition},460$
entropy	$\mathtt{assign},469$
${\tt random\_device},907$	category, 469
eof	${ t clear},469$
${\tt basic\_ios},980$	constructor, 468
epptr	message, 469
basic_streambuf, 990	operator bool, $469$
epsilon	operator!=, $470$
numeric_limits, 430	operator<, $469$
eq	operator=, 469
char_traits, 629-632	operator==, 470
equal, 849	value, 469
istreambuf_iterator, 833	error_type, 1060, 1061
equal_range, 863	exception
equal_to, 547	bad_function_call, 553
equivalent	bad_weak_ptr, 523
error_category, 464, 465	<pre><exception>, 447</exception></pre>
erase	exception, 448
deque, 737	constructor, 448
list, 749	destructor, 448
basic_string, 626	exception_ptr, 450
vector, 757	exceptions
erase_after	basic ios, 981
forward list, 742	exchange
erased	atomic type, 1108
forward list, 742	exit, 59, 61, 129, 408, 438, 444
erf, 956	EXIT_FAILURE, 438
erfc, 956	EXIT_SUCCESS, 438
errc, 460	exp, 944, 956
error_category, 460, 463	complex, 885
default_error_condition, 464	exp2, 956
equivalent, 464, 465	expired
message, $464$	weak_ptr, 534
name, 464	expm1, 956
operator!=, 464	exponential_distribution, 916
operator<, 464	constructor, 917
operator==, 464	lambda, 917
error_code, 460, 465, 468	extreme_value_distribution, 919
01101_0000, 100, 100, 100	ondiemo_varue_distribution, 919

a, 919	find, 847
b, 920	basic_string, 629
constructor, 919	map, 766
constructor, 313	± /
facet	$\begin{array}{c} \texttt{multimap}, 770 \\ \texttt{find\_end}, 847 \end{array}$
locale, 650	<del>-</del>
fail	find_first_not_of
basic_ios, 980	basic_string, 631, 632
failed	find_first_of, 848
ostreambuf_iterator, 835	basic_string, 630
failure	find_if, 847
ios_base::failure, 969	find_if_not, 847
falsename	find_last_not_of
numpunct, 680	basic_string, 632
fclose, 1036	find_last_of
_1	basic_string, 631
fdim, 956 FE_ALL_EXCEPT, 875	fisher_f_distribution, 923
	constructor, 924
FE_DTUDYZEDO 875	m, 924
FE_DUNHARD 875	n, 924
FE_DOWNWARD, 875	$\mathtt{fixed},983$
FE_INEXACT, 875	flag_type
FE_INVALID, 875	$\mathtt{basic\_regex},1069$
FE_OVERFLOW, 875	flags
FE_TONEAREST, 875	$\mathtt{ios\_base},658,971$
FE_TOWARDZERO, 875	flip
FE_UNDERFLOW, 875	$\mathtt{bitset}, 495$
FE_UPWARD, 875	$\mathtt{bitset}, 495$
feclearexcept, 875	vector bool>, $760$
fegetenv, 875	${ t float\_denorm\_style,427,434}$
fegetexceptflag, 875	${\tt numeric\_limits},432$
fegetround, 875	$float\_round\_style, 427, 433$
feholdexcept, 875	floor, $956$
fenv_t, 875	$FLT_DIG, 436$
feraiseexcept, 875	FLT_EPSILON, 436
fesetenv, 875	FLT_EVAL_METHOD, 436
fesetexceptflag, 875	FLT_MANT_DIG, 436
fesetround, 875	$FLT_MAX, 436$
fetch_	FLT_MAX_10_EXP, 436
atomic type, 1110	FLT_MAX_EXP, 436
fetestexcept, 875	FLT_MIN, 436
feupdateenv, 875	FLT_MIN_10_EXP, 436
fexcept_t, 875	FLT_MIN_EXP, 436
filebuf, 961, 1032	FLT_RADIX, 436
fill, 854	FLT_ROUNDS, 436
array, 732	flush, 971, 998, 1011, 1012, 1017
basic_ios, 978	basic_ostream, 1016
gslice_array, 949	fma, 956
indirect_array, 952	fmax, 956
mask_array, 950	fmin, 956
slice_array, 946	fmtflags
fill_n, 854	ios, 1017
<del>-</del> /	105, 1017

i h 000 071	form both a
ios_base, 969, 971	from_bytes
fopen, 1035	wstring_convert, 655
for_each, 847	from_time_t, 592
format	front
match_results, 1080, 1081	basic_string, 621
format_default, 1058	forward_list, 741
format_default, 1059	front_insert_iterator, 821
format_first_only, 1058, 1086	front_insert_iterator, 821
format_first_only, 1060	front_inserter, 822
format_no_copy, 1058, 1086	fseek, 1035
format_no_copy, 1060	fstream>, $1032$
$format\_sed, 1058$	fstream, 961
$format\_sed, 1059$	function, $553$
forward, $475$	${\tt assign}, 556$
forward_as_tuple, 487	bool conversion, 557
forward_iterator_tag, 813	destructor, $556$
<forward_list>, 729</forward_list>	function, $555$
forward_list	invocation, 557
assign, 741	operator!=, $557$
before_begin, 741	$\mathtt{operator}(), 557$
cbefore_begin, 741	operator=, $556$
clear, 743	operator==, 557
emplace_after, 742	swap, 556, 557
emplace_front, 741	target, 557
erase_after, 742	target_type, 557
erased, 742	<functional>, 539</functional>
forward_list, 740	future
front, 741	constructor, 1150, 1151
insert_after, 741, 742	get, 1151
merge, 744	operator=, 1151
pop, 741	share, 1151
push_front, 741	valid, 1152
remove, 744	wait, 1152
remove_if, 744	wait_for, 1152
resize, 743	wait_until, 1152
reverse, 745	future_category, 1145
,	
sort, 744	future_errc
splice_after, 743	make_error_code, 1145
swap, 745	make_error_condition, 1145
unique, 744	future_error
fpclassify, 958	code, 1146
fpos, 966, 974, 975	what, $1146$
state, 975	gamma digtaibution 017
frac_digits	gamma_distribution, 917
moneypunct, 693	alpha, 918
free, 512	beta, 918
freeze	constructor, 918
ostrstream, 1212	gbump
strstream, 1213	basic_streambuf, 990
${\tt strstreambuf},1207$	gcount
frexp, 956	$ exttt{basic_istream}, 1002$

25.1	150 151
generate, 854	getenv, 453, 454
$\mathtt{seed\_seq},908$	getline
generate_canonical, 909	$\texttt{basic\_istream},1004,1005$
generate_n, 854	$\mathtt{basic\_string},638,639$
$generic\_category, 463, 465$	getloc,1063
geometric_distribution, 914	$\mathtt{basic\_regex},1069$
constructor, 914	basic_streambuf, 988
p, 914	${ t ios\_base},972$
get	global
array, 732, 733	locale, $653$
auto_ptr, 1220	good
basic_istream, 1002, 1003, 1007	basic_ios, 980
future, 1151	gptr
messages, 695	basic_streambuf, 990
money_get, 689	greater, 547
num_get, 671	greater_equal, 548
pair, 480	grouping
reference_wrapper, 545	
shared_future, 1154	moneypunct, 693
	numpunct, 680
shared_ptr, 529	gslice, 946
time_get, 684	constructor, 948
tuple, 488	gslice_array, 948
unique_ptr, 519	hardware_concurrency
get_allocator	thread, 1121
basic_string, 629	has_denorm_loss
match_results, 1081	
get_date	numeric_limits, 432
time_get, 684	has_facet
get_deleter	locale, 653
shared_ptr, 533	has_infinity
$\verb"unique_ptr", 519$	numeric_limits, 431
get_future	has_quiet_NaN
${ t packaged\_task,1159}$	numeric_limits, 431
promise, 1148	has_signaling_NaN
get_id	numeric_limits, 431
${ t this\_thread}, 1122$	hash, 470, 539, 558, 600, 641
thread, $1121$	collate, $682$
get_money, 1019	$\mathtt{hash\_code},497$
get_monthname	${\tt type\_info},446$
$time_get, 684$	$\verb"type_index", 600"$
get_new_handler, 420	hex, 983
get_pointer_safety, 504	$\mathtt{hexfloat},983$
get_temporary_buffer, 510	hypot, 956
get_terminate, 420	
get_time, 1020	id
time_get, 684	$\verb locale , 650 $
get_unexpected, 420	idxl
get_weekday	operator>, $587$
time_get, 684	$\mathtt{ifstream},961,1032$
get_year	ignore
time_get, 684	$ exttt{basic_istream},1005$
51m5_500, 001	

ilogb,956	insert_iterator, 822
imag, 886	insert_iterator, 823
complex, 880, 883	inserter, 823
imbue, 1063	int16_t, 436
basic_filebuf, 1039	int32_t, 436
basic_ios, 978	int64_t, 436
basic_regex, 1069	int8_t, 436
basic_streambuf, 990	int_fast16_t, 436
ios_base, 972	int_fast32_t, 436
in	int_fast64_t, 436
codecvt, 667	int_fast8_t, 436
in_avail	int_least16_t, 436
basic_streambuf, 988	int_least32_t, 436
includes, 865	int_least64_t, 436
independent_bits_engine, 903	int_least8_t, 436
indirect_array, 950	INT_MAX, 436
operator[], 951	INT_MIN, 436
infinity	int_type
numeric_limits, 432	char_traits, 603
Init	wstring_convert, 656
ios_base::Init, 971	internal, $982$
init	intervals
basic_ios, 977, 997, 1010	piecewise_constant_distribution, 928
<pre><initializer_list>, 452</initializer_list></pre>	piecewise_linear_distribution, 930
initializer_list, 452	$\frac{1}{1}$ intmax_t, $\frac{436}{1}$
begin, $453$	intptr_t, 436
end, $453$	invalid_argument, 456, 457, 492, 493
initializer_list, 452	invalid_argument, 457
size, $453$	INVOKE, 543
inner_allocator	<iomanip>, 995</iomanip>
scoped_allocator_adaptor, 596	<ios>, 965</ios>
inner_allocator_type	ios, 961, 966
scoped_allocator_adaptor, 595	ios_base, 966
inner_product, 953	destructor, 974
inplace_merge, 864	fmtflags, 971
input_iterator_tag, 813	ios_base, 974
emplace	iostate, 969
deque, 736	precision, $972$
insert	setf, 972
deque, 736	streamsize, 972
list, 749	ios_base::failure, 969
$\mathtt{basic\_string},624,625$	ios_base::Init, 971
$\mathtt{map}, 766$	destructor, 971
multimap, 770	<iosfwd $>$ , $961$
vector, 757	iostate
push_back	${\tt ios\_base},969$
deque, 736	<iostream $>$ , $963$
push_front	${\tt iostream\_category},984$
deque, 736	iota, 955
insert_after	is
$\texttt{forward\_list},741,742$	ctype, 660

ctype <char>, <math>664</math></char>	${ t istream\_iterator,828}$
is_bind_expression, 550	constructor, 829
is_bounded	destructor, 829
numeric_limits, 433	operator!=, 830
is_error_code_enum, 460	$\mathtt{operator*}, 829$
is_error_condition_enum, 460	operator++, 829, 830
is_exact	operator->, <u>829</u>
numeric_limits, 430	operator==, 830
is_heap, 868	istreambuf_iterator, 831
is_heap_until, 868	constructor, 833
is_iec559	operator++, <u>833</u>
numeric_limits, 432	istringstream, 961, 1021
is_integer	istrstream, 1210
numeric_limits, 430	constructor, 1210
is_modulo	istrstream, 1210
numeric_limits, 433	isunordered, 958
is_open	isupper, $653^{'}$
basic_filebuf, 1035, 1046	isxdigit, 653
basic_ifstream, 1041	iter_swap, $852$
basic_ofstream, 1043	<pre><iterator>, 808</iterator></pre>
is_partitioned, 858	iword
is_permutation, 849	ios_base, 973
is_signed	_ ,
numeric_limits, 430	${\tt jmp\_buf},454$
is_sorted, 861	join
is_sorted_until, 861	thread, 1120
isalnum, 653	joinable
isalpha, 653	thread, 1120
isblank, 653	
iscntrl, 653	kill_dependency, $1101$
isctype	$\mathtt{knuth\_b},906$
regex_traits, 1063	
regular expression traits, 1094	lambda
isdigit, 653	exponential_distribution, 917
isfinite, 958	LDBL_DIG, 436
isgraph, 653	LDBL_EPSILON, 436
isgreater, 958	LDBL_MANT_DIG, 436
isgreaterequal, 958	LDBL_MAX, 436
isinf, 958	LDBL_MAX_10_EXP, 436
isless, $958$	LDBL_MAX_EXP, 436
islessequal, 958	LDBL_MIN, 436
islessgreater, 958	LDBL_MIN_10_EXP, 436
islower, 653	LDBL_MIN_EXP, 436
isnan, 958	left, 982
isnormal, 958	length
<iso646.h>, 1199</iso646.h>	char_traits, 617, 619, 624, 635
isprint, 653	basic_string, 619
ispunct, 653	codecvt, 667
isspace, 653	match_results, 1079
<pre><istream>, 994</istream></pre>	${\tt regex\_traits},1062$
istream, 961, 994	$\mathtt{sub\_match},1070$
······································	

valarray, 940	logb, 956
length_error, 456, 458, 611	logic_error, 456
length_error, 458	logic_error, 457
less, 548	logical_and, 548
less_equal, 548	logical_not, 549
lexicographical_compare, 871	logical_or, 548
lgamma, 956	_
<pre>climits&gt;, 427</pre>	lognormal_distribution, 921
	constructor, 921 m, 921
linear_congruential_engine, 898 constructor, 898	s, 921
<pre><li>&lt;1ist&gt;, 729</li></pre>	LONG_MAX, 436
list, 745	longjmp, 453, 454
assign, 748	lookup_classname
list, 748	regex_traits, 1062
splice, 750	regular expression traits, 1094
swap, 752	lookup_collatename
LLONG_MAX, 436	regex_traits, 1062
LLONG_MIN, 436	regular expression traits, 1094
llrint, 956	lower_bound, 862
11round, 956	lowest
load	numeric_limits, 429
atomic type, 1108	lrint, 956
<pre><locale>, 645, 646</locale></pre>	1round, $956$
locale, 1063, 1069, 1093	_
category, 648	m
classic, $653$	fisher_f_distribution, 924
combine, 652	lognormal_distribution, 921
constructor, 651, 652	make_error_code, 460, 467, 984
destructor, 652	future_errc, 1145
$\mathtt{facet},650$	make_error_condition, 460, 469, 984
global,653	future_errc, 1145
$\mathtt{has\_facet},653$	make_exception_ptr, 451
$\mathtt{id},650$	make_heap, 868
$\mathtt{name},\ 652$	make_move_iterator, 828
operator!=, $652$	$\mathtt{make\_pair},479$
operator(), $652$	make_ready_at_thread_exit
operator=, $652$	$ t packaged_task, 1159$
operators==, $652$	${\tt make\_shared},530$
${\tt use\_facet},653$	$\mathtt{make\_tuple},486$
lock, 1134	malloc, 512, 1201
unique_lock, 1132	map>, $760$
$\mathtt{weak\_ptr},535$	$\mathtt{map},762$
lock_guard	constructor, 765
constructor, 1129	$\mathtt{find}, 766$
destructor, 1129	insert, 766
$\log, 944, 956$	$\mathtt{map},765$
complex, 885	operator<, $765$
log10, 944, 956	operator==, $765$
complex, 885	$\mathtt{swap},767$
log1p, 956	mark_count
log2, 956	$basic\_regex, 1069$
<u> </u>	

mask_array, 949	max_exponent
operator[], 950	numeric_limits, 431
match_any, 1058	max_exponent10
match_any, 1059	numeric_limits, 431
match_continuous, 1058, 1089	max_length
match_continuous, 1059	codecvt, 667
match_default, 1058	max_size
match_flag_type, 1058, 1059, 1094	allocator, 509
match_not_bol, 1058	allocator_traits, 507
match_not_bol, 1059	array, 730
match_not_bow, 1058	basic_string, 620
match_not_bow, 1059	match_results, 1079
match_not_eol, 1058	scoped_allocator_adaptor, 596
match_not_eol, 1059	$\mathtt{MB\_LEN\_MAX},\ 436$
match_not_eow, 1058	mean
match_not_eow, 1059	normal_distribution, 920
match_not_null, 1058, 1089	poisson_distribution, 916
match_not_null, 1059	student_t_distribution, 925
match_prev_avail, 1058, 1089	$\mathtt{mem\_fn}, 552$
$\mathtt{match\_prev\_avail},1059$	$mem\_fun, 1215, 1216$
match_results, 1076, 1087, 1090	$\mathtt{mem\_fun1\_ref\_t}, 1215$
$\mathtt{begin},1080$	$\mathtt{mem\_fun1\_t}, 1215$
$\mathtt{empty},1079$	${\tt mem\_fun\_ref},\ 1215,\ 1217$
end, $1080$	$\mathtt{mem\_fun\_ref\_t}, 1215$
format, $1080, 1081$	$\mathtt{mem\_fun\_t}, 1215$
${ t get\_allocator}, 1081$	memchr, 641
$\mathtt{length},1079$	<pre><memory>, <math>498</math></memory></pre>
${\tt match\_results},1078$	$\mathtt{merge},864$
$\mathtt{matched},\ 1076$	list,751
$\mathtt{max\_size},1079$	${\tt forward\_list}, 744$
operator!=, $1082$	mersenne_twister_engine, 899
operator=, $1078$	constructor, 900
operator==, $1081$	message
operator[], 1079	do_close, 696
position, 1079	error_category, 464
prefix, 1080	error_code, 467
$\mathtt{size},1079$	${ t error\_condition},469$
state, 1078	messages, 694
str, 1079	close, 695
suffix, 1080	do_get, 696
swap, 1081	do_open, 695
max, 869	get, 695
duration, 586	open, 695
duration_values, 582	messages_byname, 696
numeric_limits, 429	min, 869
time_point, 590	duration, 586
valarray, 941	duration_values, $582$
max_align_t, 426, 427	numeric_limits, 429
max_digits10	time_point, 590
numeric_limits, 430	valarray, 941
max_element, 870	min_element, 870

min ownonent	onerator - 826 827
min_exponent numeric_limits, 430	operator-, 826, 827 operator-=, 827
min_exponent10	operator -, 827
numeric_limits, 431	operator, 826
minmax, 869, 870	operator<, 827
minmax, 809, 870 minmax_element, 870	operator<=, 827
minstd_rand, 905	operator=, 825
minstd_rand0, 905	operator==, 827
minus, 546	operator>, 827
mismatch, 849	operator>=, 827
mod, 956	operator[], 827
modf, 956	mt19937, 905
modulus, 547	mt19937_64, 905
money_get, 689	multimap, 767
do_get, 689	find, 770
get, 689 money_put, 690	insert, 770
•	multimap, 769, 770
do_put, 691	operator<, 769
put, 691	operator==, 769
moneypunct, 692	swap, 770
curr_symbol, 693	multiplies, 546 multiset, 774
decimal_point, 693	•
do_curr_symbol, 694	multiset, 776
do_decimal_point, 693	operator<, 776
do_frac_digits, 694	operator==, 776
do_grouping, 694	swap, 777
do_neg_format, 694	<pre><mutex>, 1122</mutex></pre>
do_negative_sign, 694 do_pos_format, 694	mutex
- <b>-</b>	unique_lock, 1133
do_positive_sign, 694 do_thousands_sep, 693	n
<b>-</b> _ <b>-</b> .	chi_squared_distribution, 922
frac_digits, 693 grouping, 693	fisher_f_distribution, 924
G 1 G	name
negative_sign, 693 positive_sign, 693	type_info, 446
thousands_sep, 693	error_category, 464
moneypunct_byname, 694	locale, $652$
move, 475	type_index, 600
basic_ios, 979	nan, 956
movemove, 852	narrow
move_backward, 852	basic_ios, 978
move_if_noexcept, 476	ctype, 660
move_iterator, 824	ctype <char>, 665</char>
base, 825	NDEBUG, 409
constructor, 825	nearbyint, 956
move_iterator, 825	negate, 547
operator!=, 827	negative_binomial_distribution, 914
operator*, 826	constructor, 915
operator+, 826, 827	p, 915
operator++, 826	t, 915
operator+=, 826	negative_sign
oporator., oro	5 - 5

	675
moneypunct, 693	put, 675
nested_exception, 451	<pre><numeric>, 952</numeric></pre>
nested_exception, 451	numeric_limits, 428
nested_ptr, 451	numeric_limits, 427
rethrow_if_nested, 452	denorm_min, 432
rethrow_nested, 451	digits, 429
throw_with_nested, 452	digits10, 429
nested_ptr	epsilon, 430
nested_exception, 451	float_denorm_style, 432
<new>, 439</new>	has_denorm_loss, 432
new 440 449	has_infinity, 431
operator, 440, 443	has_quiet_NaN, 431
operator, 420, 440-443, 512	has_signaling_NaN, 431
new_handler, 444	infinity, 432
next, 814	is_bounded, 433
next_permutation, 871	is_exact, 430
nextafter, 956	is_iec559, 432
nexttoward, 956	is_integer, 430
noboolalpha, 981	is_modulo, 433
none	is_signed, 430
bitset, 496	lowest, $429$
none_of, 847	$\max$ , $429$
norm, 886	$\mathtt{max\_digits10},430$
complex, 883	max_exponent, 431
normal_distribution, 920	$max_exponent10, 431$
constructor, 920	$\mathtt{min},429$
$\mathtt{mean},920$	$ exttt{min_exponent},430$
$\mathtt{stddev},921$	$\mathtt{min}\mathtt{\_exponent10},431$
noshowbase, 981	$\mathtt{quiet\_NaN},432$
noshowpoint, 981	$\mathtt{radix},430$
noshowpos, 982	$round_error, 430$
noskipws, 982	${\tt round\_style},433$
not1, 549	$\mathtt{signaling\_NaN}, 432$
not2, 550	$tinyness\_before, 433$
$\mathtt{not\_equal\_to}, 547$	$\mathtt{traps},433$
notify_all	${\tt numeric\_limits < bool>, 435}$
${\tt condition\_variable}, 1138$	$\mathtt{numpunct},679$
${\tt condition\_variable\_any}, 1142$	$\mathtt{decimal\_point}, 679$
notify_one	${\tt do\_decimal\_point}, 680$
${\tt condition\_variable}, 1137$	${ t do\_falsename}, 680$
${\tt condition\_variable\_any}, 1142$	$do\_grouping, 680$
nounitbuf, 982	$do\_thousands\_sep,680$
nouppercase, $982$	do_truename, 680
nth_element, 862	falsename, 680
NULL, 426, 427	grouping, $680$
nullptr_t, 426, 427	$\verb thousands_sep , 680$
num_get, 670	truename, 680
do_get, 671, 674	numpunct_byname, 681
get, 671	
num_put, 674	oct,983
do_put, 675, 678	off_type
- <del>-</del>	

char_traits, 603	regex_iterator, 1088
offsetof, 426, 427, 1201	regex_token_iterator, 1092
ofstream, 961, 1032	reverse_iterator, 819
once_flag, 1134	scoped_allocator_adaptor, 598
open	shared_ptr, 531
basic_filebuf, 1035, 1046	stack, 801
basic_ifstream, 1041	sub_match, 1071-1075
basic_ofstream, 1043, 1044	thread::id, 1119
messages, 695	time_point, 590
openmode	tuple, 489
ios_base, 969	type_index, 599
operator @=	unique_ptr, 522
atomic type, 1110	valarray, 943
operator auto_ptr	operator()
auto_ptr_ref, 1220	default_delete, 514, 515
operator basic_string	function, 557
sub_match, 1070	locale, $652$
operator bool	packaged_task, 1159
basic_istream, 999	random_device, 907
basic_ios, 980	reference_wrapper, 545
basic_ostream, 1012	operator*
error_code, 467	auto_ptr, 1220
error_condition, 469	back_insert_iterator, 821
shared_ptr, 530	complex, 882
unique_lock, 1133	duration, 586
unique_ptr, 519	front_insert_iterator, 822
operator C	insert_iterator, 823
atomic type, 1108	istream_iterator, 829
operator T&	istreambuf_iterator, 833
reference_wrapper, 545	move_iterator, 826
operator!	ostream_iterator, 831
basic_ios, 980	ostreambuf_iterator, 834
valarray, 939	raw_storage_iterator, 510
operator!=, 474	regex_iterator, 1089
pair, 479	regex_token_iterator, 1092
type_info, 445	reverse_iterator, 817
allocator, 509	shared_ptr, 529
basic_string, 636	unique_ptr, 519
bitset, 496	valarray, 942
complex, 882	operator*=
duration, 587	complex, 881
error_category, 464	duration, 585
error_code, 470	gslice_array, 949
error_condition, 470	indirect_array, 951
function, 557	mask_array, 950
istream_iterator, 830	slice_array, 946
istreambuf_iterator, 833	valarray, 940
locale, 652	operator+
match_results, 1082	basic_string, 633-635
move_iterator, 827	complex, 881
queue, 795	duration, 584, 590
quouo, 100	daraoron, 001, 000

000 007	
move_iterator, 826, 827	istream_iterator, 829
reverse_iterator, 818, 819	move_iterator, 826
time_point, 590	regex_iterator, 1089
$\mathtt{valarray},939,942$	regex_token_iterator, 1092
operator++	reverse_iterator, 817
atomic type, 1110	shared_ptr, 529
back_insert_iterator, 821	$\verb"unique_ptr", 519"$
duration, 584	operator
front_insert_iterator, 822	atomic type, 1110, 1111
${\tt insert\_iterator},823$	duration, $585$
${ t istream\_iterator},829,830$	${ t move\_iterator},826$
istreambuf_iterator, 833	reverse_iterator, 817
$ exttt{move\_iterator}, 826$	operator/
${\tt ostream\_iterator}, 831$	complex, 882
${\tt ostreambuf\_iterator},834$	$\mathtt{duration},586,587$
${ t raw\_storage\_iterator}, 510$	valarray, $942$
${\tt regex\_iterator},1089$	operator/=
$regex\_token\_iterator, 1092, 1093$	complex, $881$
reverse_iterator, 817	$\mathtt{duration}, 585$
operator+=	${\tt gslice\_array},949$
$\mathtt{basic\_string}, 621, 622$	${\tt indirect\_array},951$
complex, 880, 881	${\tt mask\_array},950$
duration, 585	${ t slice\_array}, 946$
${\tt gslice\_array},949$	valarray, 940
${\tt indirect\_array},951$	operator<
${\tt mask\_array},950$	$\mathtt{pair},479$
${\tt move\_iterator}, 826$	basic_string, 636
reverse_iterator, 818	duration, $587$
slice_array, 946	error_category, $464$
${\tt time\_point}, 589$	${ t error\_code},467$
valarray, 940	$error\_condition, 469$
operator-	move_iterator, 827
complex, 882	queue, $795$
$\overline{\mathtt{duration}}, 584, 590$	reverse_iterator, 818
$move_iterator, 826, 827$	shared_ptr, 531
reverse_iterator, 818, 819	stack, 801
time_point, 590	sub_match, 1071, 1073-1075
valarray, 939, 942	thread::id, 1119
operator-=	time_point, 590
complex, 881	tuple, 489
duration, 585	type_index, 599
gslice_array, 949	unique_ptr, 522, 523
indirect_array, 951	valarray, 943
mask_array, 950	operator«
move iterator, 827	shared_ptr, 533
reverse_iterator, 818	sub_match, 1076
slice_array, 946	operator<<
time_point, 589	bitset, 496, 497
valarray, 940	complex, 883
operator->	operator<<=
auto_ptr, 1220	bitset, 493
<b>-1</b> ,	· · · · · · · · · · · · · · · · · · ·

operator<=, 474	$\mathtt{basic\_stringbuf},1023$
$\mathtt{pair},479$	${\tt basic\_stringstream},1031$
$\mathtt{basic\_string},637$	$\verb"enable_shared_from_this", 536$
$\mathtt{duration},587$	${ t error\_code},466$
${\tt move\_iterator},827$	${ t error\_condition},469$
queue, $795$	exception, $448$
reverse_iterator, 819	$front_insert_iterator, 822$
$\mathtt{shared\_ptr}, 522, 531$	function, $556$
stack, 801	future, 1151
sub_match, 1071-1076	gslice_array, 949
thread::id, 1119	indirect_array, 951
time_point, 590	insert_iterator, 823
tuple, 489	$\frac{1}{1}$
type_index, 599	mask_array, 950
unique_ptr, 523	match_results, 1078
valarray, 943	move_iterator, 825
operator<<	ostream_iterator, 831
basic_istream, 1008	ostreambuf_iterator, 834
basic_ostream, 1012, 1014, 1015, 1017	packaged_task, 1158
basic_string, 638	pair, 478
error_code, 467	promise, 1148
thread::id, 1119	raw_storage_iterator, 510
valarray, 942	reference_wrapper, 545
operator<<=	shared_future, 1154
gslice_array, 949	shared_ptr, 528, 529
	slice_array, 946
indirect_array, 951	_ • • ·
mask_array, 950	thread, 1120
slice_array, 946	tuple, 485, 486
valarray, 940	unique_lock, 1132
operator=	unique_ptr, 518, 519
bad_alloc, 444	valarray, 936, 937, 942
bad_cast, 446	weak_ptr, 534
bad_exception, 449	operator==
bad_typeid, 447	pair, 479
reverse_iterator, 817	type_info, 445
atomic type, 1107	allocator, 509
auto_ptr, 1219	basic_string, 635
auto_ptr_ref, 1220	bitset, 495
back_insert_iterator, 820	complex, 882
basic_filebuf, 1035	duration, 587
basic_fstream, 1045	error_category, 464
basic_ifstream, 1041	error_code, 470
basic_iostream, 1008	error_condition, 470
basic_istream, 998	function, $557$
basic_istringstream, 1028	istream_iterator, 830
basic_ofstream, 1043	istreambuf_iterator, 833
${\tt basic\_ostream}, 1011$	$\mathtt{match\_results},1081$
${\tt basic\_ostringstream},1029$	${ t move\_iterator}, 827$
${\tt basic\_regex},1067$	queue, $795$
${\tt basic\_streambuf},989$	${ t regex\_iterator},1088$
$\mathtt{basic\_string}, 618, 619$	${\tt regex\_token\_iterator},1090,1092$

reverse_iterator, 818	$\mathtt{istream},9991002$
${ t scoped\_allocator\_adaptor,} \ 598$	$ exttt{valarray},942$
${ t shared\_ptr}, 531$	operator>>=
$\mathtt{stack},801$	${\sf gslice\_array},949$
$\mathtt{sub\_match},10711075$	indirect_array, 951
thread::id, 1118	mask_array, 950
time_point, 590	slice_array, 946
tuple, 489	valarray, 940
type_index, 599	operator[]
unique_ptr, 522	basic_string, 621
valarray, $943$	bitset, 496
operator>, 474	indirect_array, 951
pair, 479	$\mathtt{map}, 765$
basic_string, 636, 637	mask_array, 950
idx1, 587	match_results, 1079
move_iterator, 827	move_iterator, 827
queue, 796	reverse_iterator, 818
reverse_iterator, 819	unique_ptr, 521
shared_ptr, 531	unordered_map, 782
stack, 802	unordered_multimap, 786
sub_match, 1071-1075	valarray, 937-939
thread::id, 1119	operator%
time_point, 591	duration, 587
tuple, 489	·
type_index, 600	valarray, 942 operator%=
	<del>-</del>
unique_ptr, $522$ , $523$ valarray, $943$	duration, 585
- · · · · · · · · · · · · · · · · · · ·	gslice_array, 949
operator>=, 474	indirect_array, 951
pair, 479	mask_array, 950
basic_string, 637	slice_array, 946
duration, 588	valarray, 940
move_iterator, 827	operator&
queue, 796	bitset, 497
reverse_iterator, 819	valarray, 942
shared_ptr, 531	operator&=
stack, 802	bitset, 493
sub_match, 1071-1076	gslice_array, 949
thread::id, 1119	indirect_array, 951
time_point, 591	mask_array, 950
tuple, 490	slice_array, 946
type_index, 600	valarray, 940
unique_ptr, 522, 523	operator&&
valarray, 943	valarray, 942, 943
operator>>	operator^
bitset, 496, 497	bitset, 497
complex, 882	valarray, 942
operator>>=	operator^=
$\mathtt{bitset}, 494$	bitset, 493
operator>>	gslice_array, 949
basic_istream, 1000	indirect_array, 951
$\mathtt{basic\_string},638$	$\mathtt{mask\_array},950$

${ t slice\_array},946$	$\verb"unique_lock", 1133"$
valarray, 940	
operator~	p
bitset, 494	bernoulli_distribution, 913
valarray, 939	${\tt binomial\_distribution},913$
operators==	${\tt geometric\_distribution},914$
locale, $652$	negative_binomial_distribution, 915
operator	packaged_task
bitset, 497	constructor, 1158
valarray, 942	destructor, 1158
operator =	$\mathtt{get\_future}, 1159$
bitset, 493	make_ready_at_thread_exit, 1159
gslice_array, 949	operator(), 1159
indirect_array, 951	operator=, 1158
mask_array, 950	reset, 1159
slice_array, 946	swap, 1158, 1160
valarray, 940	valid, 1158
operator	pair, 476, 484, 486
valarray, 942, 943	get, 480
<pre></pre>	operator=, 478
	pair, 477, 478
ostream, 961, 995	swap, 478
ostream_iterator, 830	param
constructor, 831	seed_seq, 909
destructor, 831	partial_sort, 860
operator*, 831	partial_sort_copy, 861
operator++, 831	
operator=, 831	partial_sum, 954
ostreambuf_iterator, 834	partition, 858
constructor, 834	partition_copy, 858
ostringstream, 961, 1021	partition_point, 859
ostrstream, 1211	pbackfail
constructor, 1211	basic_filebuf, 1037
ostrstream, 1211	basic_streambuf, 993
out	basic_stringbuf, 1024
$\mathtt{codecvt},666$	strstreambuf, 1208
out_of_range, 458, 492-496, 611	pbase
$\mathtt{out\_of\_range},458$	basic_streambuf, 990
out_of_range_error, 456	pbump
outer_allocator	${\tt basic\_streambuf}, 990$
${ t scoped\_allocator\_adaptor,596}$	pcount
output_iterator_tag, 813	$\mathtt{ostrstream},1212$
overflow	$\mathtt{strstream},1213$
$basic_filebuf, 1037$	$\mathtt{strstreambuf},1207$
basic_streambuf, 993	peek
$\mathtt{basic\_stringbuf},1025$	${\tt basic\_istream},1005$
strstreambuf, 1207	${\tt piecewise\_constant\_distribution},927$
overflow_error, 456, 459, 460, 492, 495	constructor, 927, 928
overflow_error, 459	densities, 928
owner_before	intervals, 928
shared_ptr, 530, 535	piecewise_construct, 480
owns_lock	piecewise_construct_t, 480
- ··	= ·

niocovice linear distribution 020	complay 884
piecewise_linear_distribution, 929 constructor, 929, 930	complex, 884 promise
densities, 930	constructor, 1148
intervals, 930	destructor, 1148
placeholders, 552	get_future, 1148
——————————————————————————————————————	<b>-</b> - · · ·
plus, 546	operator=, 1148
pointer	set_exception, 1149
allocator_traits, 506	set_exception_at_thread_exit, 1149
pointer_to	set_value, 1149
pointer_traits, 503	set_value_at_thread_exit, 1149
pointer_to_binary_function, 1214	swap, 1148, 1150
pointer_to_unary_function, 1214	propagate_on_container_copy_assignment
pointer_traits, 502	allocator_traits, 506
difference_type, 502	scoped_allocator_adaptor, 595
element_type, 502	propagate_on_container_move_assignment
pointer_to, 503	allocator_traits, 507
rebind, $502$	scoped_allocator_adaptor, 595
poisson_distribution, 915	propagate_on_container_swap
constructor, 916	allocator_traits, 507
mean, 916	${ t scoped\_allocator\_adaptor,} \ 595$
polar	proxy
complex, 884	$istreambuf\_iterator, 832$
pop	ptr_fun, 1214
${ t priority\_queue},799$	$\mathtt{ptrdiff\_t}, 426$
forward_list, 741	pubimbue
pop_back	basic_streambuf, 987
$\mathtt{basic\_string}, 626$	pubseekoff
pop_heap, 867	${\tt basic\_streambuf},988$
pos_type	pubseekpos
$\mathtt{char\_traits}, 603$	${\tt basic\_streambuf},988$
position	pubsetbuf
$\mathtt{match\_results},1079$	${\tt basic\_streambuf},988$
positive_sign	pubsync
moneypunct, 693	${\tt basic\_streambuf},988$
pow, 886, 944, 956	push
complex, 885	$priority_queue, 798$
pptr	push_back
${\tt basic\_streambuf}, 990$	$basic_string, 623$
precision	push_front
$\mathtt{ios\_base},658,972$	forward_list, 741
prefix	push_heap, 867
match_results, 1080	put
prev, 814	basic_ostream, 1016
prev_permutation, 872	money_put, 691
priority_queue, 796	num_put, 675
emplace, 799	time_put, 687
priority_queue, 797	put_money, 1019
swap, 799	put_time, 1020
probabilities	putback
discrete_distribution, 926	basic_istream, 1006
proj	putenv, 453
	1

pword	wbuffer_convert, 658
ios_base, 974	rdstate
<queue>, 793</queue>	basic_ios, 980
queue, 793	read
swap, 796	basic_istream, 1005 readsome
quick_exit, 438, 439	
quiet_NaN	basic_istream, 1005
numeric_limits, 432	real, 886
namorio_rimios, for	complex, 880, 883
radix	realloc, 512, 1201 rebind
numeric_limits, 430	
raise, 454	pointer_traits, 502
<random>, 895-897</random>	rebind_alloc
random_access_iterator_tag, 813	allocator_traits, 507
random_device, 906	
constructor, 907	reference_wrapper, 546 reference_wrapper, 544
entropy, $907$	
operator(), 907	cref, 546
random_shuffle, 857	get, 545
range_error, 456, 459	operator T&, 545
range_error, 459	operator(), 545
ranlux24, 906	operator=, 545
ranlux24_base, 905	ref, 546
ranlux48, 906	reference_wrapper, 545
ranlux48_base, 905	<regex>, 1051</regex>
ratio, 575	regex, 1051
ratio_equal, 577	regex_constants, 1057
ratio_greater, 577	error_type, 1060, 1061 match_flag_type, 1058
ratio_greater_equal, 577	syntax_option_type, 1057
ratio_less, 577	regex_error, 1061, 1064, 1094
ratio_less_equal, 577	constructor, 1061
ratio_not_equal, 577	regex_iterator, 1087
raw_storage_iterator	increment, 1089
constructor, 510	operator!=, 1088
operator*, $510$	operator*, 1089
operator++, 510	operator++, 1089
operator=, 510	operator->, 1089
rbegin	operator==, 1088
basic_string, 619	regex_iterator, 1088
rdbuf	regex_match, 1082, 1083
${\tt basic\_filebuf},1046$	regex_replace, 1085, 1086
${\tt basic\_ifstream},1041$	regex_search, 1084, 1085
basic_ios, 978	regex_token_iterator, 1089
${\tt basic\_istringstream},1028$	end-of-sequence, 1090
basic_ofstream, 1043	operator!=, 1092
${\tt basic\_ostringstream},1030$	operator*, 1092
$\verb basic_stringstream , 1032 $	operator++, 1092, 1093
istrstream, 1211	operator->, 1092
$\mathtt{ostrstream}, 1212$	operator==, 1090, 1092
strstream, 1213	regex_token_iterator, 1091

1001	1017
regex_traits, 1061	resetiosflags, 1017
char_class_type, 1062	resize
isctype, 1063	deque, 736
length, 1062	list, 749
lookup_classname, 1062	basic_string, 620
lookup_collatename, 1062	forward_list, 743
transform, $1062$	valarray, 941
$transform_primary, 1062$	vector, 756
translate, $1062$	$rethrow_exception, 451$
translate_nocase, 1062	rethrow_if_nested
value, 1063	nested_exception, 452
register_callback	rethrow_nested
ios_base, 974	${\tt nested\_exception},451$
regular expression traits	return_temporary_buffer, $510$
isctype, $1094$	reverse, $856$
$lookup\_classname, 1094$	$\mathtt{list}, 752$
${\tt lookup\_collatename},1094$	${\tt forward\_list},  745$
${\tt transform\_primary},1094$	reverse_copy, $856$
rel_ops, 472	$reverse\_iterator, 815$
release	${ t reverse\_iterator}, 816$
auto_ptr, 1220	$\mathtt{base},817$
unique_lock, 1133	constructor, 816
${\tt unique\_ptr}, 520$	operator++, $817$
remainder, 956	operator, 817
remove, 854	rfind
list, 751	$basic_string, 630$
forward_list, 744	right, 982
remove_copy, 855	rint, 956
remove_copy_if, 855	rotate, 856
remove_if, 854	rotate_copy, 857
forward_list, 744	round, 956
remquo, 956	round_error
rend	numeric_limits, 430
basic_string, 619	round_indeterminate, 434
rep	round_style
system_clock, 591	numeric_limits, 433
replace, 853	round_to_nearest, 434
basic_string, 626-628	round_toward_infinity, 434
replace_copy, 853	round_toward_neg_infinity, 434
replace_copy_if, 853	round_toward_zero, 434
replace_if, 853	runtime_error, 456, 458
reserve	$\texttt{runtime\_error},459$
basic_string, 620	s
vector, 756	lognormal_distribution, 921
reset	sbumpc
auto_ptr, 1220	<del>-</del>
bitset, 494	basic_streambuf, 988
packaged_task, 1159	scalbn, 956
shared_ptr, 529	scalbn, 956
unique_ptr, 520, 521	scan_is
$\mathtt{weak\_ptr},534$	$\mathtt{ctype}, 660$

ctype <char>, <math>664</math></char>	${\tt scoped\_allocator\_adaptor}, 598$
scan_not	sentry
ctype, 660	${\tt basic\_istream},998$
ctype < char > , $664$	${\tt basic\_ostream},1011$
SCHAR_MAX, 436	constructor, 998, 1011
SCHAR_MIN, 436	$\langle \mathtt{set} \rangle$ , $761$
scientific, 983	set, 771
scoped_allocator_adaptor	$\mathtt{bitset}, 494$
allocate, $596$	operator<, 773
$\mathtt{construct}, 597, 598$	operator==, 773
constructor, 595, 596	set, 773
$ ext{deallocate}, 596$	swap, 773
destructor, 598	set_difference, 866
inner_allocator, 596	set_exception
inner_allocator_type, 595	promise, 1149
max_size, 596	set_exception_at_thread_exit
operator!=, 598	promise, 1149
operator==, 598	set_intersection, 866
outer_allocator, 596	set_new_handler, 420, 445
propagate_on_container_copy_assignment,	set_rdbuf
595	basic_ios, 979
propagate_on_container_move_assignment,	set_symmetric_difference, 866
595	set_terminate, 420, 449
propagate_on_container_swap, 595	set_unexpected, 420, 1221
select_on_container_copy_construction,	set_union, 865
598	set_value
search, 850	promise, 1149
search_n, 850	set_value_at_thread_exit
seed_seq	promise, 1149
constructor, 908	setbase, $1018$
generate, 908	setbuf
param, 909	basic_filebuf, 1038
size, 909	basic_streambuf, 991, 1026
seekdir	streambuf, 1210
ios_base, 969	strstreambuf, 1210
seekg	setenv, $453$
basic_istream, 1006	setf
seekoff	ios_base, 971, 972
basic_filebuf, 1038	setfill, 1018
basic_streambuf, 991	setg
basic_stringbuf, 1025	basic_streambuf, 990
strstreambuf, 1208	strstreambuf, 1207
seekp	setiosflags, 1017
basic_ostream, 1012	setjmp, 419, 454
seekpos	<pre><setjmp.h>, 453</setjmp.h></pre>
basic_filebuf, 1039	setlocale, 406
basic_streambuf, 991	setp
basic_stringbuf, 1025	basic_streambuf, 990
strstreambuf, 1209	setprecision, 1018
select_on_container_copy_construction	setstate
allocator_traits, 507	
allocator_traits, our	$\mathtt{basic\_ios},980$

setw, 1019	$\mathtt{shared\_ptr}, 526$
sgetc	static_pointer_cast, 532
basic streambuf, 988	swap, 529, 532
	unique, 530
sgetn	<del>-</del>
basic_streambuf, 988 share	use_count, 530 shift
future, 1151	valarray, 941 showbase, 981
shared_from_this	,
enable_shared_from_this, 536 shared future	showmanyc basic_filebuf, 1036
constructor, 1153	basic_streambuf, 991, 1036
destructor, 1153	showpoint, 981
get, 1154	showpos, 981
operator=, 1154	shrink_to_fit
valid, 1154	basic_string, 620
wait, 1155	deque, 736
wait, 1155 wait_for, 1155	vector, 756
wait_until, 1155	SHRT MAX, 436
shared_ptr, 524, 536	SHRT_MIN, 436
atomic_compare_exchange_strong, 538	shuffle, 857
atomic_compare_exchange_strong	shuffle_order_engine, 904
explicit, 538	constructor, 905
atomic_compare_exchange_weak, 538	sig_atomic_t, 454
atomic_compare_exchange_weak	SIG_DFL, 454
explicit, 538	SIG_ERR, 454
atomic_exchange, 538	SIG_IGN, 454
atomic_exchange_explicit, 538	SIGABRT, 454
atomic_is_lock_free, 537	SIGFPE, 454
atomic_load, 537	SIGILL, 454
atomic_load_explicit, 537	SIGINT, 454
atomic_store, 537	signal, 454
atomic_store_explicit, 537	<pre><signal.h>, 453</signal.h></pre>
const_pointer_cast, 532	signaling_NaN
constructor, 526–528	numeric_limits, 432
destructor, 528	signbit, 958
dynamic_pointer_cast, 532	SIGSEGV, 454
get, 529	SIGTERM, 454
get_deleter, 533	sin, 944, 956
operator bool, 530	complex, 885
operator!=, 531	sinh, 944, 956
operator*, 529	complex, 885
operator->, 529	size
operator<, 531	array, 730, 732
operator«, 533	basic_string, 619
operator<=, 522, 531	bitset, 495
operator=, 528, 529	gslice, 948
operator=, 531	initializer_list, 453
operator>, 531	match_results, 1079
operator>=, 531	seed_seq, 909
owner_before, 530, 535	slice, 945
reset, 529	size_t, 106, 426
	,,

size_type	${\tt stdarg.h>},453$
${\tt allocator\_traits},506$	${\sf stdbool.h>},454$
$\mathtt{skipws},982$	stddev
sleep_for	$normal_distribution, 921$
$this\_thread, 1122$	<stdexcept $>$ , $456$
sleep_until	<stdlib.h>, 453, 1202</stdlib.h>
this_thread, 1122	stod,640
slice, 944	stof, 639, 640
slice,945	stoi, 639, 640
$slice\_array, 945$	stol, 639, 640
snextc	stold, 639, 640
basic_streambuf, 988	stoll, 639, 640
sort, 860	store
list, 752	atomic type, $1107$
forward_list, 744	stoul, 639, 640
sort_heap, 868	stoull, 639, 640
splice	str
list, 750	basic_istringstream, 1028
list, 750	basic_ostringstream, 1030
splice_after	basic_stringbuf, 1024
forward_list, 743	basic_stringstream, 1032
sputbackc	istrstream, 1211
basic_streambuf, 989	match_results, 1079
sputc	ostrstream, 1212
basic_streambuf, 989	strstream, 1213
sputn	strstreambuf, 1207
basic_streambuf, 989	$\mathtt{sub\_match}, 10\overset{'}{70}$
sqrt, 944, 956	strchr, 641
complex, 885	<pre><streambuf>, 984</streambuf></pre>
<pre><sstream>, 1021</sstream></pre>	streambuf, 961, 984
<staarg.h>, 453</staarg.h>	streamoff, 966, 975, 1203
stable_partition, 858	streamsize, 966
stable_sort, 860	ios_base, 972
<stack>, 799</stack>	strftime, 688
stack, 799	stride
swap, 802	gslice,948
start	slice, $945$
gslice, $948$	<string>, 607</string>
slice, $945$	stringbuf, 961, 1021
state	stringstream, 961
fpos, $975$	strlen, 1206, 1211
match_results, 1078	strpbrk, 641
wbuffer_convert, 658	strrchr, 641
wstring_convert, 656	strstr, 641
state_type	strstream, 1212
char_traits, 603	destructor, 1213
wbuffer_convert, 658	strstream, 1213
wstring_convert, 656	strstreambuf, 1204, 1206
static_pointer_cast	strstreambuf, 1205
shared_ptr, 532	destructor, 1207
<pre><stdalign.h>, 454</stdalign.h></pre>	setg, 1207
<b>o</b> , -	· · · · · · · · · · · · · · · · · ·

student_t_distribution, 924	multiset, 777
constructor, 925	packaged_task, 1158, 1160
mean, 925	pair, 478
sub_match, 1069	priority_queue, 799
compare, 1070	promise, 1148, 1150
constructor, 1070	queue, 796
length, 1070	set, 773
operator basic_string, 1070	shared_ptr, 529, 532
operator!=, 1071-1075	stack, 802
operator<, 1071-1075	thread, 1120, 1121
operator«, 1076	tuple, 486
operator<=, 1071-1076	unique_lock, 1133
operator==, 1071-1075	unique_ptr, 520
operator>, 1071-1075	unordered_map, 782
operator>=, 1071-1076	unordered_multimap, 786
str,1070	unordered_multiset, 792
substr	${\tt unordered\_set}, 789$
$\mathtt{basic\_string}, 632$	$\mathtt{valarray},940,944$
${\tt subtract\_with\_carry\_engine},900$	$\mathtt{vector}, 756, 757$
constructor, 901	vector <bool>, 760</bool>
suffix	$\mathtt{weak\_ptr},\ 534,\ 535$
$\mathtt{match\_results},1080$	swap(unique_ptr&, unique_ptr&), $522$
sum	${ t swap\_ranges}, 852$
valarray, 941	sync
sungetc	$ exttt{basic_filebuf}, 1039$
basic_streambuf, 989	$\mathtt{basic\_istream},1006$
swap, 474, 490	basic_streambuf, 991
pair, 479	sync_with_stdio
array, 731, 732	ios_base, 973
basic_filebuf, 1035	syntax_option_type, 1057, 1058
basic_fstream, 1045	$\mathtt{awk},1058$
basic_ifstream, 1041	basic, $1058$
basic_ios, 979	collate, 1058, 1094
basic_iostream, 1008	ECMAScript, 1058
basic_istream, 998	egrep, 1058
basic_istringstream, 1028	extended, $1058$
basic_ofstream, 1043	grep, 1058
basic_ostream, 1011	icase, 1058
basic_ostringstream, 1030	nosubs, 1058
basic_regex, 1069	optimize, 1058
basic_streambuf, 989	syntax_option_type
basic_string, 629, 637	awk, 1058
basic_stringbuf, 1024	basic, 1058
basic_stringstream, 1031, 1032	collate, 1058
deque, 737	ECMAScript, 1058
forward_list, 745	egrep, 1058
function, 556, 557	extended, 1058
list, 752	grep, 1058
map, 767	icase, 1058
match_results, 1081	nosubs, 1058
multimap, 770	optimize, $1058$

system, 453, 454	constructor, 1118
$system\_category, 463, 465$	operator!=, $1119$
system_clock	operator<, $1119$
rep, 591	operator $<=$ , $1119$
system_error, 460, 470	operator $<<$ , $1119$
code, 471	operator==, 1118
system_error, 471	operator>, 1119
what, 471	operator>=, 1119
,	throw_with_nested
t	nested_exception, 452
binomial_distribution, 913	tie, 487
negative_binomial_distribution, 915	basic_ios, 977, 978
table	time, 453
ctype <char>, 665</char>	<time.h>, 453</time.h>
tan, 944, 956	
complex, 886	time_get, 682
tanh, 944, 956	date_order, 684
	do_date_order, 685
complex, 886	do_get, 686
target	do_get_date, 685
function, 557	do_get_monthname, 685
target_type	${ t do_{ t get\_time},685}$
function, 557	$\mathtt{do\_get\_weekday}, 685$
tellg	$\mathtt{do\_get\_year}, 686$
basic_istream, 1006	$\mathtt{get}, 684$
tellp	$\mathtt{get\_date},684$
$\mathtt{basic\_ostream},1012$	$\mathtt{get\_monthname},684$
terminate, 438, 439, 449, 450, 1221	$\mathtt{get\_time},684$
terminate_handler, 420, 449	get_weekday, 684
test	get_year, 684
$\mathtt{bitset},496$	time_get_byname, 687
tgamma, 956	time_point
this_thread	constructor, 589
get_id, 1122	$\max, 590$
sleep_for, 1122	min, 590
sleep_until, 1122	operator!=, 590
yield, 1122	operator+, 590
thousands_sep	operator+=, 589
moneypunct, 693	operator-, 590
numpunct, 680	operator-=, 589
<pre><thread>, 1117</thread></pre>	operator<, 590
thread	<u>-</u>
constructor, 1119, 1120	operator<=, 590
destructor, 1120	operator==, 590
detach, 1121	operator>, 591
•	operator>=, 591
get_id, 1121	time_since_epoch, 589
hardware_concurrency, 1121	time_point_cast, 591
join, 1120	time_put, 687
joinable, 1120	do_put, 688
operator=, 1120	$\mathtt{put},687$
swap, 1120, 1121	time_put_byname, 688
thread::id	time_since_epoch

time_point, 589	operator>, 489
tinyness_before	operator>=, 490
numeric_limits, 433	swap, 486
to_bytes	tie, 487
wstring_convert, 656	tuple, 483
to_string, 640	tuple_cat, 487
bitset, 495	tuple_element, 480, 488, 732
to_time_t, 591	tuple_size, 480, 488, 732
	type_index
to_ullong bitset, 495	constructor, 599
to ulong	hash_code, 600
bitset, 495	name, 600
to_wstring, 640	operator!=, 599
tolower, 654	operator<, 599
	operator<=, 599
ctype, 660 ctype <char>, 664</char>	<u>-</u>
	operator==, 599
toupper, 654	operator>, 600
ctype, 660	operator>=, 600
ctype <char>, 664 transform, 853</char>	type_info, 99, 445 type_info::name
collate, 682	implementation-defined, 446
•	· ·
regex_traits, 1062	<typeinfo>, 445, 599</typeinfo>
transform_primary	UCHAR_MAX, 436
regex_traits, 1062 translate	uflow
	basic_filebuf, 1037
regex_traits, 1062	basic_streambuf, 993
translate_nocase	uint16_t, 436
regex_traits, 1062	uint32_t, 436
traps	uint64_t, 436
numeric_limits, 433 treat_as_floating_point, 581	uint8_t, 436
	uint_fast16_t, 436
truename	uint_fast32_t, 436
numpunct, 680	uint_fast64_t, 436
trunc, 956	uint_fast8_t, 436
try_lock, 1134 unique_lock, 1132	uint_least16_t, 436
try_lock_for	uint_least32_t, 436
unique_lock, 1133	uint_least64_t, 436
try_lock_until	uint_least8_t, 436
• = -	UINT_MAX, 436
unique_lock, 1132 <tuple>, 481</tuple>	uintmax_t, 436
tuple, 480, 482, 732	uintptr_t, 436
constructor, 483, 484	ULLONG_MAX, 436
forward_as_tuple, 487	unary_function, 544, 1213
get, 488	unary_negate, 549
make_tuple, 486	uncaught_exception, 450
operator!=, 489	undeclare_no_pointers, 503
operator: -, 489 operator<, 489	undeclare_reachable, 503
operator<, 489 operator<=, 489	underflow
operator, 489 operator, 485, 486	basic_filebuf, 1037
operator=, 489, 480 operator==, 489	basic_streambuf, 992
operator, 403	

basic_stringbuf, 1024	operator==, 522
strstreambuf, 1208	operator>, 522, 523
underflow_error, 456	operator>=, 522, 523
underflow_error, 460	operator[], 521
unexpected, 1221	release, 520
unexpected_handler, 420, 1221	reset, $520, 521$
unget	$\mathtt{swap},520$
$\mathtt{basic\_istream},1006$	$\verb"unique_ptr", 516"$
uniform_int_distribution, 910	$\mathtt{unitbuf},982$
a, 911	unlock
b, 911	$\mathtt{unique\_lock}, 1133$
constructor, 911	<pre><unordered_map>, 777</unordered_map></pre>
$uniform\_real\_distribution, 911$	$\mathtt{unordered\_map},\ 777779$
<b>a</b> , 912	$\mathtt{at},782$
b, 912	$ exttt{operator[]},782$
constructor, 911	$\mathtt{swap},782$
uninitialized_copy, 511	${\tt unordered\_map},\ 781$
uninitialized_copy_n, 511	unordered_multimap, 777, 783
uninitialized_fill, $511$	$ exttt{operator[]},786$
uninitialized_fill_n, 512	$\mathtt{swap},786$
unique, $855$	unordered_multimap, 785
list,751	unordered_multiset, 778, 789, 790
$forward_list, 744$	$\mathtt{swap},792$
$\mathtt{shared\_ptr},530$	${\tt unordered\_multiset}, 792$
unique_copy, 855	<pre><unordered_set>, 778</unordered_set></pre>
unique_lock	$\mathtt{unordered\_set}, 778, 786$
constructor, 1130–1132	$\mathtt{swap}, \overline{789}$
destructor, 1132	unordered_set, 789
lock, $1132$	unsetf
mutex, 1133	${ t ios\_base},972$
operator bool, 1133	unshift
operator=, 1132	codecvt, $667$
owns_lock, 1133	upper_bound, 863
release, 1133	uppercase, 982
swap, 1133	use_count
try_lock, 1132	shared_ptr, 530
try_lock_for, 1133	$\mathtt{weak\_ptr}, 534$
try_lock_until, 1132	use_facet
unlock, 1133	locale, 653
unique_ptr, 528	uses_allocator, 504, 1148, 1160
constructor, 517, 518, 521	uses_allocator <tuple>, 490</tuple>
destructor, 518	USHRT_MAX, 436
get, 519	<pre><utility>, 472</utility></pre>
get_deleter, 519	vacinty, 172
operator bool, 519	va_arg, 454
operator!=, 522	va_copy, 454
operator*, 519	va_end, 419, 454
operator->, 519	va_list, 419, 454
operator<, 522, 523	va_start, 453, 454
operator<, 523	<pre><valarray>, 931</valarray></pre>
operator=, 528, 519	valarray, 933, 948
opolasor , 510, 610	<b>,</b>

begin, 952	swap, 760
constructor, 935, 936	void_pointer
destructor, 936	allocator_traits, 506
end, $952$	d1100d001_01d100; 000
operator!=, 943	wait
operator*, 942	condition_variable, 1138
operator*=, 940	condition_variable_any, 1142
operator+, 942	future, 1152
operator+=, 940	shared_future, 1155
operator-=, 940	wait_for
operator/, 942	condition_variable, 1139, 1140
operator/=, 940	condition_variable_any, 1143
operator<, 943	future, 1152
operator<=, 943	shared_future, 1155
operator<<, 942	wait_until
operator<<=, 940	condition_variable, 1138, 1139
operator=, 936, 937, 942	condition_variable_any, 1142, 1143
operator==, 943	future, 1152
operator>, 943	$\mathtt{shared\_future},1155$
operator>=, 943	wbuffer_convert, 657
operator>>, 942	constructor, 658
operator>>=, 940	destructor, 658
operator%, 942	rdbuf, 658
operator%=, 940	state,658
operator&, 942	state_type, 658
operator&=, 940	wcerr, $965$
operator&&, 943	wcin, 964
operator, 942	wclog, 965
operator^=, 940	wcout, 964
operator, 942	wcschr, $642$
operator =, 940	wcspbrk, 642
operator  , 943	wcsrchr, 642
swap, 940, 944	wcsstr, 642
valarray, 935	$\mathtt{weak\_ptr}, 527, 533$
valid	constructor, 534
future, $1152$	destructor, 534
packaged_task, 1158	expired, $534$
shared_future, 1154	$\mathtt{lock},535$
value	operator=, $534$
error_code, 467	$\mathtt{reset},534$
error_condition, 469	$\mathtt{swap},534,535$
regex_traits, 1063	${\tt use\_count},534$
<pre><vector>, 729</vector></pre>	weibull_distribution, $918$
vector, 752	<b>a</b> , 919
operator<, 754	b, 919
operator==, 754	constructor, 918
vector, 754, 755	wfilebuf, $961$ , $1032$
assign, 755	${\tt wfstream},961$
swap, 757	what
vector <bool>, 757</bool>	bad_alloc, 444
flip, 760	$\mathtt{bad\_cast}, 446$

$\mathtt{bad}$ _exception, $449$	zero
$\mathtt{bad\_typeid},447$	$\mathtt{duration},586$
exception, $448$	${\tt duration\_values}, 581$
$\mathtt{bad\_weak\_ptr},524$	
$future\_error, 1146$	
$system_error, 471$	
wide_string	
wstring_convert, $656$	
widen	
$\mathtt{basic\_ios},978$	
ctype, 660	
ctype <char>, <math>664</math></char>	
width	
$\mathtt{ios\_base},658,972$	
wifstream,961,1032	
wios, $966$	
wistream, $961$ , $994$	
wistringstream, $961$ , $1021$	
wmemchr, 642	
$\texttt{wofstream},\ 961,\ 1032$	
wostream,961,995	
wostringstream, $961$ , $1021$	
wregex, 1051	
write	
${\tt basic\_ostream},1016$	
ws, 1000, 1007	
wstreambuf, 961, 984	
wstring_convert, 654	
byte_string, $655$	
constructor, $656$	
converted, $655$	
destructor, 657	
${ t from\_bytes}, 655$	
$\mathtt{int\_type},656$	
state,656	
$\mathtt{state\_type}, 656$	
to_bytes, $656$	
wide_string, $656$	
wstringbuf, 961, 1021	
wstringstream, 961	
11	
xalloc	
ios_base, 973	
xsgetn	
basic_streambuf, 991	
xsputn	
basic_streambuf, 993	
yield	
this_thread, 1122	

© ISO/IEC N3337

## Index of implementation-defined behavior

The entries in this section are rough descriptions; exact specifications are at the indicated page in the general text.

```
dynamic initialization of thread-local objects be-
#pragma, 396
                                                                  fore entry, 61
additional formats for time_get::do_get_date,
                                                         effect of calling basic_filebuf::setbuf with non-
alignment, 75
                                                                  zero arguments, 1038
alignment additional values, 76
                                                         effect of calling basic filebuf::sync when a get
alignment of bit-fields within a class object, 216
                                                                  area exists, 1039
allocation of bit-fields within a class object, 216
                                                         effect of calling basic_streambuf::setbuf with
argument values to construct basic_ios::failure,
                                                                  non-zero arguments, 1026
                                                         effect of calling ios_base::sync_with_stdio af-
assignability of placeholder objects, 552
                                                                  ter any input or output operation on stan-
                                                                  dard streams, 973
behavior of attribute scoped token, 168
                                                         effect on C locale of calling locale::global, 653
behavior of iostream classes when traits::pos -
                                                         encoding of universal character name not in exe-
         type is not streampos or when traits::off_-
                                                                  cution character set, 26
         type is not streamoff, 961
                                                         error_category for errors originating outside the
behavior of non-standard attributes, 168
                                                                  operating system, 424
bits in a byte, 6
                                                         exception type when shared_ptr constructor fails,
                                                                  526 - 528
choice of larger or smaller value of floating literal,
                                                         exceptions thrown by standard library functions
                                                                  that do not have an exception specifica-
concatenation of some types of string literals, 29
                                                                  tion, 424
conversions between pointers and integers, 101
                                                         execution character-set and execution wide-character
converting characters from source character set to
                                                                  set, 18
        execution character set, 17
                                                         exit status, 439
converting function pointer to object pointer and
                                                         extended signed integer types, 71
         vice versa, 102
                                                         extensions to enum type launch, 1145
default number of buckets in unordered map, 781
                                                         formatted character sequence generated by time_-
default number of buckets in unordered multimap,
                                                                  put::do_put in C locale, 688
default number of buckets in unordered_multiset,
                                                         headers for freestanding implementation, 408
                                                         interactive device, 8
default number of buckets in unordered_set, 789
defining main in freestanding environment, 58
                                                         linkage of main, 59
definition and meaning of __STDC__, 396
                                                         linkage of names from Standard C library, 409
definition and meaning of __STDC_VERSION__-
                                                         locale names, 651
         , 397
derived type for typeid, 99
                                                         manner of search for included source file, 389
diagnostic message, 2
                                                         mapping from name to catalog when calling messages
distinctness of string literals, 28
                                                                  ::do_open, 695
dynamic initialization of static objects before main,
                                                         mapping header name to header or external source
         60
                                                                  file, 21
```

mapping physical source file characters to basic source character set, 16	start-up and termination in freestanding environ- ment, 58
mapping to message when calling messages::do	string resulting fromfunc, 188
get, 696	support for extended alignment, 574
meaning of asm declaration, 164	support for over-aligned types, 106, 509, 510
meaning of attribute declaration, 134	support for over-angled types, 100, 303, 310 supported multibyte character encoding rules, 604,
meaning of attribute declaration, 134	606
negative value of character literal in preprocessor,	
389	text of $\_\_DATE\_\_$ when date of translation is not
nesting limit for #include directives, 390	available, 396
number of threads in a program under a freestand-	text of $\_\_TIME\_\_$ when time of translation is not
ing implementation, 11	available, 396
numeric values of character literals in #if direc-	type of ios_base::streamoff, 1203
tives, 389	type of ios_base::streampos, 1203
	type of ptrdiff_t, 114, 427
parameters to main, 58	type of regex_constants::error_type, 1060
passing argument of class type through ellipsis, 95	type of size_t, 427
physical source file characters, 16	type of streamoff, 604
presence and meaning of native_handle_type and	type of streampos, 604
native_handle, 1114	type of u16streampos, 605
	type of u32streampos, 606
rank of extended signed integer type, 82	type of wstreampos, 606
representation of char, 71	type of array::const_iterator, 730
required libraries for freestanding implementation,	type of array::iterator, 730
5	, F = =====, +==========================
result of exception::what, 448	underlying source of random numbers for random
result of inexact floating-point conversion, 80	$\mathtt{shuffle},858$
result of right shift of negative value, 114	underlying type for enumeration, 151
return value of bad_alloc::what, 444	use of non-POF function as signal handler, 454
return value of bad_cast::what, 446	
return value of bad_exception::what, 449	value of ctype <char>::table_size, 663</char>
return value of bad_typeid::what, 447	value of character literal outside range of corre-
return value of char_traits <char16_t>::eof, 605</char16_t>	sponding type, $26$
return value of char_traits <char32_t>::eof, 606</char32_t>	value of multicharacter literal, 25
return value of type_info::name(), 446	value of result of inexact integer to floating-point conversion, 80
search locations for "" header, 390	value of result of unsigned to signed conversion, 80
search locations for <> header, 389	value of wide-character literal containing multiple
semantics of linkage specification on templates,	characters, 25
303	value of wide-character literal with single c-char
semantics of linkage specifiers, 165	that is not in execution wide-character
semantics of non-standard escape sequences, 26	set, $25$
sequence of places searched for a header, 389	value representation of floating-point types, 72
set of blank characters defined by regex_traits	value representation of pointer types, 73
::isctype, $1063$	values of a trivially copyable type, 69
signedness of char, 71	values of various ATOMICLOCK_FREE macros,
signedness of plain integral bit-field, 216	1101
sizeof applied to fundamental types other than	-
char, signed char, and unsigned char,	whether get_pointer_safety returns pointer
105	<pre>safety::relaxed or pointer_safety::preferred</pre>
stack unwinding before call to std::terminate(),	if the implementation has relaxed pointer
381, 386	safety, 504

© ISO/IEC N3337

```
whether {\tt time\_get::do\_get\_year} accepts two-digit
        year numbers, 686
whether an implementation has relaxed or strict
        pointer safety, 65
whether locale object is global or per-thread, 648
whether sequence pointers are copied by basic_-
        filebuf move constructor, 1034
whether sequence pointers are copied by basic_-
         stringbuf move constructor, 1023
whether source of translation units must be avail-
        able to locate template definitions, 17
whether stack is unwound before calling std::terminate()
        when a noexcept specification is violated,
        386
whether values are rounded or truncated to the
        required precision when converting be-
        tween time_t values and time_point ob-
        jects., 591, 592
which functions in Standard C++ library may be
        recursively reentered, 423
```