

INTERNATIONAL STANDARD

ISO/IEC
14882

First edition
1998-09-01

Programming languages — C++

Langages de programmation — C++



Contents

1	General	1
1.1	Scope	1
1.2	Normative references	1
1.3	Definitions	1
1.3.1	argument	1
1.3.2	diagnostic message	2
1.3.3	dynamic type	2
1.3.4	ill-formed program	2
1.3.5	implementation-defined behavior	2
1.3.6	implementation limits	2
1.3.7	locale-specific behavior	2
1.3.8	multibyte character	2
1.3.9	parameter	2
1.3.10	signature	2
1.3.11	static type	2
1.3.12	undefined behavior	2
1.3.13	unspecified behavior	3
1.3.14	well-formed program	3
1.4	Implementation compliance	3
1.5	Structure of this International Standard	4
1.6	Syntax notation	4
1.7	The C++ memory model	4
1.8	The C++ object model	
1.9	Program execution	5

© ISO/IEC 1998

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

1.10	Acknowledgments	8
2	Lexical conventions	9
2.1	Phases of translation	9
2.2	Character sets.....	10
2.3	Trigraph sequences	11
2.4	Preprocessing tokens	11
2.5	Alternative tokens.....	12
2.6	Tokens.....	12
2.7	Comments.....	12
2.8	Header names.....	13
2.9	Preprocessing numbers	13
2.10	Identifiers.....	13
2.11	Keywords.....	14
2.12	Operators and punctuators	15
2.13	Literals	15
2.13.1	Integer literals	15
2.13.2	Character literals.....	16
2.13.3	Floating literals	18
2.13.4	String literals.....	19
2.13.5	Boolean literals	19
3	Basic concepts	21
3.1	Declarations and definitions	21
3.2	One definition rule.....	22
3.3	Declarative regions and scopes.....	24
3.3.1	Point of declaration.....	25
3.3.2	Local scope	26
3.3.3	Function prototype scope.....	26
3.3.4	Function scope	27
3.3.5	Namespace scope.....	27
3.3.6	Class scope.....	27
3.3.7	Name hiding.....	28
3.4	Name lookup.....	29
3.4.1	Unqualified name lookup	29
3.4.2	Argument-dependent name lookup.....	32
3.4.3	Qualified name lookup	33

3.4.3.1	Class members	35
3.4.3.2	Namespace members	35
3.4.4	Elaborated type specifiers	38
3.4.5	Class member access	39
3.4.6	Using-directives and namespace aliases	40
3.5	Program and linkage	41
3.6	Start and termination.....	43
3.6.1	Main function.....	43
3.6.2	Initialization of non-local objects	44
3.6.3	Termination.....	45
3.7	Storage duration.....	45
3.7.1	Static storage duration	46
3.7.2	Automatic storage duration.....	46
3.7.3	Dynamic storage duration.....	46
3.7.3.1	Allocation functions.....	47
3.7.3.2	Deallocation functions	47
3.7.4	Duration of sub-objects.....	48
3.8	Object Lifetime	48
3.9	Types.....	51
3.9.1	Fundamental types	53
3.9.2	Compound types	54
3.9.3	CV-qualifiers	55
3.10	Lvalues and rvalues	55
4	Standard conversions	57
4.1	Lvalue-to-rvalue conversion	57
4.2	Array-to-pointer conversion	58
4.3	Function-to-pointer conversion	58
4.4	Qualification conversions	58
4.5	Integral promotions.....	59
4.6	Floating point promotion	59
4.7	Integral conversions.....	60
4.8	Floating point conversions.....	60
4.9	Floating-integral conversions	60
4.10	Pointer conversions.....	60
4.11	Pointer to member conversions	61

4.12	Boolean conversions	61
5	Expressions	63
5.1	Primary expressions	64
5.2	Postfix expressions	66
5.2.1	Subscripting	66
5.2.2	Function call	66
5.2.3	Explicit type conversion (functional notation)	68
5.2.4	Pseudo destructor call	68
5.2.5	Class member access	68
5.2.6	Increment and decrement	69
5.2.7	Dynamic cast	70
5.2.8	Type identification	71
5.2.9	Static cast	72
5.2.10	Reinterpret cast	73
5.2.11	Const cast	74
5.3	Unary expressions	76
5.3.1	Unary operators	76
5.3.2	Increment and decrement	77
5.3.3	Sizeof	77
5.3.4	New	78
5.3.5	Delete	81
5.4	Explicit type conversion (cast notation)	82
5.5	Pointer-to-member operators	83
5.6	Multiplicative operators	83
5.7	Additive operators	84
5.8	Shift operators	85
5.9	Relational operators	85
5.10	Equality operators	86
5.11	Bitwise AND operator	87
5.12	Bitwise exclusive OR operator	87
5.13	Bitwise inclusive OR operator	87
5.14	Logical AND operator	87
5.15	Logical OR operator	88
5.16	Conditional operator	88
5.17	Assignment operators	89

5.18	Comma operator	90
5.19	Constant expressions	90
6	Statements.....	93
6.1	Labeled statement	93
6.2	Expression statement	93
6.3	Compound statement or block.....	93
6.4	Selection statements.....	94
6.4.1	The <code>if</code> statement.....	95
6.4.2	The <code>switch</code> statement	95
6.5	Iteration statements.....	95
6.5.1	The <code>while</code> statement.....	96
6.5.2	The <code>do</code> statement.....	96
6.5.3	The <code>for</code> statement.....	97
6.6	Jump statements.....	97
6.6.1	The <code>break</code> statement.....	97
6.6.2	The <code>continue</code> statement.....	98
6.6.3	The <code>return</code> statement	98
6.6.4	The <code>goto</code> statement	98
6.7	Declaration statement	98
6.8	Ambiguity resolution.....	99
7	Declarations	101
7.1	Specifiers	102
7.1.1	Storage class specifiers.....	103
7.1.2	Function specifiers.....	104
7.1.3	The <code>typedef</code> specifier.....	105
7.1.4	The <code>friend</code> specifier.....	106
7.1.5	Type specifiers.....	106
7.1.5.1	The <i>cv-qualifiers</i>	107
7.1.5.2	Simple type specifiers.....	108
7.1.5.3	Elaborated type specifiers.....	109
7.2	Enumeration declarations	110
7.3	Namespaces	112
7.3.1	Namespace definition	112
7.3.1.1	Unnamed namespaces.....	113
7.3.1.2	Namespace member definitions.....	113
7.3.2	Namespace alias.....	115
7.3.3	The <code>using</code> declaration	115
7.3.4	Using directive.....	120
7.4	The <code>asm</code> declaration	123

7.5	Linkage specifications	123
8	Declarators	127
8.1	Type names	128
8.2	Ambiguity resolution	128
8.3	Meaning of declarators	130
8.3.1	Pointers	131
8.3.2	References	132
8.3.3	Pointers to members	133
8.3.4	Arrays	133
8.3.5	Functions	135
8.3.6	Default arguments	137
8.4	Function definitions	140
8.5	Initializers	141
8.5.1	Aggregates	144
8.5.2	Character arrays	146
8.5.3	References	147
9	Classes	149
9.1	Class names	149
9.2	Class members	151
9.3	Member functions	153
9.3.1	Nonstatic member functions	154
9.3.2	The <code>this</code> pointer	155
9.4	Static members	156
9.4.1	Static member functions	157
9.4.2	Static data members	157
9.5	Unions	158
9.6	Bit-fields	159
9.7	Nested class declarations	160
9.8	Local class declarations	161
9.9	Nested type names	161
10	Derived classes	163
10.1	Multiple base classes	164
10.2	Member name lookup	165
10.3	Virtual functions	168

10.4	Abstract classes.....	172
11	Member access control	175
11.1	Access specifiers.....	176
11.2	Accessibility of base classes and base class members.....	177
11.3	Access declarations.....	178
11.4	Friends	179
11.5	Protected member access	182
11.6	Access to virtual functions.....	183
11.7	Multiple access	183
11.8	Nested classes	184
12	Special member functions.....	185
12.1	Constructors.....	185
12.2	Temporary objects	187
12.3	Conversions	188
12.3.1	Conversion by constructor.....	189
12.3.2	Conversion functions	190
12.4	Destructors	191
12.5	Free store	194
12.6	Initialization.....	195
12.6.1	Explicit initialization	196
12.6.2	Initializing bases and members.....	197
12.7	Construction and destruction	200
12.8	Copying class objects	203
13	Overloading	209
13.1	Overloadable declarations.....	209
13.2	Declaration matching.....	211
13.3	Overload resolution	212
13.3.1	Candidate functions and argument lists	213
13.3.1.1	Function call syntax.....	214
13.3.1.1.1	Call to named function.....	214
13.3.1.1.2	Call to object of class type.....	215
13.3.1.2	Operators in expressions.....	216

13.3.1.3	Initialization by constructor	218
13.3.1.4	Copy-initialization of class by user-defined conversion.....	218
13.3.1.5	Initialization by conversion function	218
13.3.1.6	Initialization by conversion function for direct reference binding	219
13.3.2	Viable functions.....	219
13.3.3	Best Viable Function	219
13.3.3.1	Implicit conversion sequences	221
13.3.3.1.1	Standard conversion sequences	222
13.3.3.1.2	User-defined conversion sequences.....	223
13.3.3.1.3	Ellipsis conversion sequences.....	223
13.3.3.1.4	Reference binding.....	224
13.3.3.2	Ranking implicit conversion sequences.....	224
13.4	Address of overloaded function.....	226
13.5	Overloaded operators.....	227
13.5.1	Unary operators.....	228
13.5.2	Binary operators.....	229
13.5.3	Assignment	229
13.5.4	Function call	229
13.5.5	Subscripting.....	230
13.5.6	Class member access	230
13.5.7	Increment and decrement.....	230
13.6	Built-in operators	231
14	Templates.....	235
14.1	Template parameters.....	236
14.2	Names of template specializations.....	238
14.3	Template arguments.....	239
14.3.1	Template type arguments.....	241
14.3.2	Template non-type arguments	242
14.3.3	Template template arguments.....	243
14.4	Type equivalence	244
14.5	Template declarations	244
14.5.1	Class templates	244
14.5.1.1	Member functions of class templates.....	245
14.5.1.2	Member classes of class templates	246
14.5.1.3	Static data members of class templates.....	246
14.5.2	Member templates	246
14.5.3	Friends	248
14.5.4	Class template partial specializations	250
14.5.4.1	Matching of class template partial specializations	252
14.5.4.2	Partial ordering of class template specializations	252
14.5.4.3	Members of class template specializations	253
14.5.5	Function templates.....	254
14.5.5.1	Function template overloading	254
14.5.5.2	Partial ordering of function templates	256

14.6	Name resolution.....	257
14.6.1	Locally declared names	260
14.6.2	Dependent names.....	262
14.6.2.1	Dependent types.....	263
14.6.2.2	Type-dependent expressions.....	263
14.6.2.3	Value-dependent expressions	264
14.6.2.4	Dependent template arguments.....	264
14.6.3	Non-dependent names.....	265
14.6.4	Dependent name resolution	265
14.6.4.1	Point of instantiation.....	265
14.6.4.2	Candidate functions	266
14.6.5	Friend names declared within a class template.....	266
14.7	Template instantiation and specialization.....	267
14.7.1	Implicit instantiation	268
14.7.2	Explicit instantiation	271
14.7.3	Explicit specialization.....	272
14.8	Function template specializations.....	277
14.8.1	Explicit template argument specification	278
14.8.2	Template argument deduction	280
14.8.2.1	Deducing template arguments from a function call.....	282
14.8.2.2	Deducing template arguments taking the address of a function template	283
14.8.2.3	Deducing conversion function template arguments.....	283
14.8.2.4	Deducing template arguments from a type	283
14.8.3	Overload resolution	288
15	Exception handling	291
15.1	Throwing an exception	292
15.2	Constructors and destructors.....	294
15.3	Handling an exception	294
15.4	Exception specifications	296
15.5	Special functions.....	298
15.5.1	The <code>terminate()</code> function	298
15.5.2	The <code>unexpected()</code> function	299
15.5.3	The <code>uncaught_exception()</code> function.....	299
15.6	Exceptions and access.....	299
16	Preprocessing directives	301
16.1	Conditional inclusion.....	302
16.2	Source file inclusion	303
16.3	Macro replacement	304
16.3.1	Argument substitution	305
16.3.2	The <code>#</code> operator	305
16.3.3	The <code>##</code> operator	306

16.3.4	Rescanning and further replacement.....	306
16.3.5	Scope of macro definitions	306
16.4	Line control.....	308
16.5	Error directive	308
16.6	Pragma directive	308
16.7	Null directive	308
16.8	Predefined macro names	309
17	Library introduction.....	311
17.1	Definitions	311
17.1.1	arbitrary-positional stream.....	311
17.1.2	character.....	311
17.1.3	character container type	311
17.1.4	comparison function	311
17.1.5	component.....	312
17.1.6	default behavior	312
17.1.7	handler function	312
17.1.8	iostream class templates	312
17.1.9	modifier function	312
17.1.10	object state	312
17.1.11	narrow-oriented iostream classes.....	312
17.1.12	NTCTS.....	312
17.1.13	observer function	312
17.1.14	replacement function.....	312
17.1.15	required behavior	312
17.1.16	repositional stream.....	313
17.1.17	reserved function.....	313
17.1.18	traits class.....	313
17.1.19	wide-oriented iostream classes	313
17.2	Additional definitions	313
17.3	Method of description (Informative)	313
17.3.1	Structure of each subclause.....	313
17.3.1.1	Summary.....	314
17.3.1.2	Requirements	314
17.3.1.3	Specifications.....	314
17.3.1.4	C Library.....	315
17.3.2	Other conventions.....	315
17.3.2.1	Type descriptions.....	315
17.3.2.1.1	Enumerated types.....	316
17.3.2.1.2	Bitmask types.....	316
17.3.2.1.3	Character sequences.....	317
17.3.2.1.3.1	Byte strings	317
17.3.2.1.3.2	Multibyte strings.....	318
17.3.2.1.3.3	Wide-character sequences.....	318
17.3.2.2	Functions within classes	318
17.3.2.3	Private members	318

17.4	Library-wide requirements	318
17.4.1	Library contents and organization	319
17.4.1.1	Library contents	319
17.4.1.2	Headers	319
17.4.1.3	Freestanding implementations	320
17.4.2	Using the library	320
17.4.2.1	Headers	320
17.4.2.2	Linkage	321
17.4.3	Constraints on programs	321
17.4.3.1	Reserved names	321
17.4.3.1.1	Macro names	321
17.4.3.1.2	Global names	321
17.4.3.1.3	External linkage	322
17.4.3.1.4	Types	322
17.4.3.2	Headers	322
17.4.3.3	Derived classes	322
17.4.3.4	Replacement functions	322
17.4.3.5	Handler functions	323
17.4.3.6	Other functions	323
17.4.3.7	Function arguments	324
17.4.3.8	Required paragraph	324
17.4.4	Conforming implementations	324
17.4.4.1	Headers	324
17.4.4.2	Restrictions on macro definitions	324
17.4.4.3	Global functions	324
17.4.4.4	Member functions	325
17.4.4.5	Reentrancy	325
17.4.4.6	Protection within classes	325
17.4.4.7	Derived classes	325
17.4.4.8	Restrictions on exception handling	325
18	Language support library	327
18.1	Types	327
18.2	Implementation properties	328
18.2.1	Numeric limits	328
18.2.1.1	Template class <code>numeric_limits</code>	328
18.2.1.2	<code>numeric_limits</code> members	329
18.2.1.3	Type <code>float_round_style</code>	333
18.2.1.4	Type <code>float_denorm_style</code>	334
18.2.1.5	<code>numeric_limits</code> specializations	334
18.2.2	C Library	335
18.3	Start and termination	336
18.4	Dynamic memory management	337
18.4.1	Storage allocation and deallocation	337
18.4.1.1	Single-object forms	337
18.4.1.2	Array forms	338
18.4.1.3	Placement forms	339
18.4.2	Storage allocation errors	340
18.4.2.1	Class <code>bad_alloc</code>	340
18.4.2.2	Type <code>new_handler</code>	340

18.4.2.3	set_new_handler	341
18.5	Type identification.....	341
18.5.1	Class type_info.....	341
18.5.2	Class bad_cast	342
18.5.3	Class bad_typeid.....	342
18.6	Exception handling	343
18.6.1	Class exception.....	343
18.6.2	Violating <i>exception-specifications</i>	344
18.6.2.1	Class bad_exception	344
18.6.2.2	Type unexpected_handler	345
18.6.2.3	set_unexpected.....	345
18.6.2.4	unexpected	345
18.6.3	Abnormal termination.....	345
18.6.3.1	Type terminate_handler	345
18.6.3.2	set_terminate.....	345
18.6.3.3	terminate.....	345
18.6.4	uncaught_exception.....	346
18.7	Other runtime support.....	346
19	Diagnostics library.....	349
19.1	Exception classes	349
19.1.1	Class logic_error	349
19.1.2	Class domain_error.....	350
19.1.3	Class invalid_argument	350
19.1.4	Class length_error.....	350
19.1.5	Class out_of_range.....	351
19.1.6	Class runtime_error	351
19.1.7	Class range_error	351
19.1.8	Class overflow_error	351
19.1.9	Class underflow_error.....	352
19.2	Assertions	352
19.3	Error numbers	352
20	General utilities library	353
20.1	Requirements	353
20.1.1	Equality comparison	353
20.1.2	Less than comparison	353
20.1.3	Copy construction.....	354
20.1.4	Default construction.....	354
20.1.5	Allocator requirements	354
20.2	Utility components.....	357
20.2.1	Operators.....	357
20.2.2	Pairs	358
20.3	Function objects.....	359
20.3.1	Base.....	361

20.3.2	Arithmetic operations	361
20.3.3	Comparisons	362
20.3.4	Logical operations	363
20.3.5	Negators	363
20.3.6	Binders	364
20.3.6.1	Template class <code>binder1st</code>	364
20.3.6.2	<code>bind1st</code>	364
20.3.6.3	Template class <code>binder2nd</code>	364
20.3.6.4	<code>bind2nd</code>	365
20.3.7	Adaptors for pointers to functions	365
20.3.8	Adaptors for pointers to members	366
20.4	Memory	368
20.4.1	The default allocator	368
20.4.1.1	allocator members	369
20.4.1.2	allocator globals	370
20.4.2	Raw storage iterator	370
20.4.3	Temporary buffers	371
20.4.4	Specialized algorithms	371
20.4.4.1	<code>uninitialized_copy</code>	371
20.4.4.2	<code>uninitialized_fill</code>	372
20.4.4.3	<code>uninitialized_fill_n</code>	372
20.4.5	Template class <code>auto_ptr</code>	372
20.4.5.1	<code>auto_ptr</code> constructors	373
20.4.5.2	<code>auto_ptr</code> members	373
20.4.5.3	<code>auto_ptr</code> conversions	374
20.4.6	C Library	374
20.5	Date and time	375
21	Strings library	377
21.1	Character traits	377
21.1.1	Character traits requirements	377
21.1.2	traits typedefs	379
21.1.3	<code>char_traits</code> specializations	379
21.1.3.1	<code>struct char_traits<char></code>	379
21.1.3.2	<code>struct char_traits<wchar_t></code>	380
21.2	String classes	381
21.3	Template class <code>basic_string</code>	383
21.3.1	<code>basic_string</code> constructors	387
21.3.2	<code>basic_string</code> iterator support	390
21.3.3	<code>basic_string</code> capacity	390
21.3.4	<code>basic_string</code> element access	391
21.3.5	<code>basic_string</code> modifiers	392
21.3.5.1	<code>basic_string::operator+=</code>	392
21.3.5.2	<code>basic_string::append</code>	392
21.3.5.3	<code>basic_string::assign</code>	393
21.3.5.4	<code>basic_string::insert</code>	393
21.3.5.5	<code>basic_string::erase</code>	394
21.3.5.6	<code>basic_string::replace</code>	395
21.3.5.7	<code>basic_string::copy</code>	396

21.3.5.8	<code>basic_string::swap</code>	397
21.3.6	<code>basic_string</code> string operations	397
21.3.6.1	<code>basic_string::find</code>	397
21.3.6.2	<code>basic_string::rfind</code>	398
21.3.6.3	<code>basic_string::find_first_of</code>	398
21.3.6.4	<code>basic_string::find_last_of</code>	399
21.3.6.5	<code>basic_string::find_first_not_of</code>	399
21.3.6.6	<code>basic_string::find_last_not_of</code>	400
21.3.6.7	<code>basic_string::substr</code>	400
21.3.6.8	<code>basic_string::compare</code>	400
21.3.7	<code>basic_string</code> non-member functions	401
21.3.7.1	<code>operator+</code>	401
21.3.7.2	<code>operator==</code>	402
21.3.7.3	<code>operator!=</code>	402
21.3.7.4	<code>operator<</code>	403
21.3.7.5	<code>operator></code>	403
21.3.7.6	<code>operator<=</code>	403
21.3.7.7	<code>operator>=</code>	404
21.3.7.8	<code>swap</code>	404
21.3.7.9	Inserters and extractors	404
21.4	Null-terminated sequence utilities	405
22	Localization library	409
22.1	Locales	409
22.1.1	Class <code>locale</code>	410
22.1.1.1	<code>locale</code> types	412
22.1.1.1.1	Type <code>locale::category</code>	412
22.1.1.1.2	Class <code>locale::facet</code>	414
22.1.1.1.3	Class <code>locale::id</code>	414
22.1.1.2	<code>locale</code> constructors and destructor	415
22.1.1.3	<code>locale</code> members	416
22.1.1.4	<code>locale</code> operators	416
22.1.1.5	<code>locale</code> static members	416
22.1.2	<code>locale</code> globals	417
22.1.3	Convenience interfaces	417
22.1.3.1	Character classification	417
22.1.3.2	Character conversions	417
22.2	Standard <code>locale</code> categories	418
22.2.1	The <code>ctype</code> category	418
22.2.1.1	Template class <code>ctype</code>	418
22.2.1.1.1	<code>ctype</code> members	419
22.2.1.1.2	<code>ctype</code> virtual functions	420
22.2.1.2	Template class <code>ctype_byname</code>	421
22.2.1.3	<code>ctype</code> specializations	422
22.2.1.3.1	<code>ctype<char></code> destructor	423
22.2.1.3.2	<code>ctype<char></code> members	423
22.2.1.3.3	<code>ctype<char></code> static members	424
22.2.1.3.4	<code>ctype<char></code> virtual functions	424
22.2.1.4	Class <code>ctype_byname<char></code>	425
22.2.1.5	Template class <code>codecvt</code>	425
22.2.1.5.1	<code>codecvt</code> members	426

22.2.1.5.2	codecvt virtual functions	427
22.2.1.6	Template class codecvt_byname	429
22.2.2	The numeric category	429
22.2.2.1	Template class num_get	429
22.2.2.1.1	num_get members	431
22.2.2.1.2	num_get virtual functions	431
22.2.2.2	Template class num_put	433
22.2.2.2.1	num_put members	434
22.2.2.2.2	num_put virtual functions	434
22.2.3	The numeric punctuation facet	437
22.2.3.1	Template class numpunct	437
22.2.3.1.1	numpunct members	438
22.2.3.1.2	numpunct virtual functions	439
22.2.3.2	Template class numpunct_byname	439
22.2.4	The collate category	439
22.2.4.1	Template class collate	439
22.2.4.1.1	collate members	440
22.2.4.1.2	collate virtual functions	440
22.2.4.2	Template class collate_byname	441
22.2.5	The time category	441
22.2.5.1	Template class time_get	441
22.2.5.1.1	time_get members	442
22.2.5.1.2	time_get virtual functions	443
22.2.5.2	Template class time_get_byname	444
22.2.5.3	Template class time_put	444
22.2.5.3.1	time_put members	445
22.2.5.3.2	time_put virtual functions	445
22.2.5.4	Template class time_put_byname	445
22.2.6	The monetary category	446
22.2.6.1	Template class money_get	446
22.2.6.1.1	money_get members	446
22.2.6.1.2	money_get virtual functions	446
22.2.6.2	Template class money_put	448
22.2.6.2.1	money_put members	448
22.2.6.2.2	money_put virtual functions	448
22.2.6.3	Template class moneypunct	449
22.2.6.3.1	moneypunct members	450
22.2.6.3.2	moneypunct virtual functions	450
22.2.6.4	Template class moneypunct_byname	451
22.2.7	The message retrieval category	452
22.2.7.1	Template class messages	452
22.2.7.1.1	messages members	452
22.2.7.1.2	messages virtual functions	453
22.2.7.2	Template class messages_byname	453
22.2.8	Program-defined facets	453
22.3	C Library Locales	457
23	Containers library	459
23.1	Container requirements	459
23.1.1	Sequences	462
23.1.2	Associative containers	464

23.2	Sequences	467
23.2.1	Template class <code>deque</code>	470
23.2.1.1	<code>deque</code> constructors, copy, and assignment	472
23.2.1.2	<code>deque</code> capacity	473
23.2.1.3	<code>deque</code> modifiers	473
23.2.1.4	<code>deque</code> specialized algorithms	473
23.2.2	Template class <code>list</code>	474
23.2.2.1	<code>list</code> constructors, copy, and assignment	476
23.2.2.2	<code>list</code> capacity	477
23.2.2.3	<code>list</code> modifiers	477
23.2.2.4	<code>list</code> operations	477
23.2.2.5	<code>list</code> specialized algorithms	479
23.2.3	Container adaptors	479
23.2.3.1	Template class <code>queue</code>	479
23.2.3.2	Template class <code>priority_queue</code>	480
23.2.3.2.1	<code>priority_queue</code> constructors	481
23.2.3.2.2	<code>priority_queue</code> members	481
23.2.3.3	Template class <code>stack</code>	481
23.2.4	Template class <code>vector</code>	482
23.2.4.1	<code>vector</code> constructors, copy, and assignment	484
23.2.4.2	<code>vector</code> capacity	485
23.2.4.3	<code>vector</code> modifiers	485
23.2.4.4	<code>vector</code> specialized algorithms	486
23.2.5	Class <code>vector<bool></code>	486
23.3	Associative containers	488
23.3.1	Template class <code>map</code>	490
23.3.1.1	<code>map</code> constructors, copy, and assignment	492
23.3.1.2	<code>map</code> element access	493
23.3.1.3	<code>map</code> operations	493
23.3.1.4	<code>map</code> specialized algorithms	493
23.3.2	Template class <code>multimap</code>	493
23.3.2.1	<code>multimap</code> constructors	496
23.3.2.2	<code>multimap</code> operations	496
23.3.2.3	<code>multimap</code> specialized algorithms	496
23.3.3	Template class <code>set</code>	496
23.3.3.1	<code>set</code> constructors, copy, and assignment	498
23.3.3.2	<code>set</code> specialized algorithms	499
23.3.4	Template class <code>multiset</code>	499
23.3.4.1	<code>multiset</code> constructors	501
23.3.4.2	<code>multiset</code> specialized algorithms	501
23.3.5	Template class <code>bitset</code>	502
23.3.5.1	<code>bitset</code> constructors	503
23.3.5.2	<code>bitset</code> members	504
23.3.5.3	<code>bitset</code> operators	506
24	Iterators library	509
24.1	Iterator requirements	509
24.1.1	Input iterators	510
24.1.2	Output iterators	511
24.1.3	Forward iterators	512
24.1.4	Bidirectional iterators	513
24.1.5	Random access iterators	513

24.2	Header <iterator> synopsis	514
24.3	Iterator primitives	516
24.3.1	Iterator traits.....	516
24.3.2	Basic iterator	517
24.3.3	Standard iterator tags	518
24.3.4	Iterator operations.....	519
24.4	Predefined iterators.....	519
24.4.1	Reverse iterators	519
24.4.1.1	Template class <code>reverse_iterator</code>	520
24.4.1.2	<code>reverse_iterator</code> requirements.....	521
24.4.1.3	<code>reverse_iterator</code> operations	521
24.4.1.3.1	<code>reverse_iterator</code> constructor.....	521
24.4.1.3.2	Conversion.....	521
24.4.1.3.3	<code>operator*</code>	521
24.4.1.3.4	<code>operator-></code>	522
24.4.1.3.5	<code>operator++</code>	522
24.4.1.3.6	<code>operator--</code>	522
24.4.1.3.7	<code>operator+</code>	522
24.4.1.3.8	<code>operator+=</code>	522
24.4.1.3.9	<code>operator-</code>	523
24.4.1.3.10	<code>operator-=</code>	523
24.4.1.3.11	<code>operator[]</code>	523
24.4.1.3.12	<code>operator==</code>	523
24.4.1.3.13	<code>operator<</code>	523
24.4.1.3.14	<code>operator!=</code>	523
24.4.1.3.15	<code>operator></code>	523
24.4.1.3.16	<code>operator>=</code>	524
24.4.1.3.17	<code>operator<=</code>	524
24.4.1.3.18	<code>operator-</code>	524
24.4.1.3.19	<code>operator+</code>	524
24.4.2	Insert iterators	524
24.4.2.1	Template class <code>back_insert_iterator</code>	525
24.4.2.2	<code>back_insert_iterator</code> operations.....	525
24.4.2.2.1	<code>back_insert_iterator</code> constructor	525
24.4.2.2.2	<code>back_insert_iterator::operator=</code>	525
24.4.2.2.3	<code>back_insert_iterator::operator*</code>	525
24.4.2.2.4	<code>back_insert_iterator::operator++</code>	525
24.4.2.2.5	<code>back_inserter</code>	526
24.4.2.3	Template class <code>front_insert_iterator</code>	526
24.4.2.4	<code>front_insert_iterator</code> operations	526
24.4.2.4.1	<code>front_insert_iterator</code> constructor.....	526
24.4.2.4.2	<code>front_insert_iterator::operator=</code>	526
24.4.2.4.3	<code>front_insert_iterator::operator*</code>	526
24.4.2.4.4	<code>front_insert_iterator::operator++</code>	527
24.4.2.4.5	<code>front_inserter</code>	527
24.4.2.5	Template class <code>insert_iterator</code>	527
24.4.2.6	<code>insert_iterator</code> operations.....	527
24.4.2.6.1	<code>insert_iterator</code> constructor	527
24.4.2.6.2	<code>insert_iterator::operator=</code>	527
24.4.2.6.3	<code>insert_iterator::operator*</code>	528
24.4.2.6.4	<code>insert_iterator::operator++</code>	528
24.4.2.6.5	<code>inserter</code>	528

24.5	Stream iterators	528
24.5.1	Template class <code>istream_iterator</code>	528
24.5.1.1	<code>istream_iterator</code> constructors and destructor	529
24.5.1.2	<code>istream_iterator</code> operations	529
24.5.2	Template class <code>ostream_iterator</code>	530
24.5.2.1	<code>ostream_iterator</code> constructors and destructor	531
24.5.2.2	<code>ostream_iterator</code> operations	531
24.5.3	Template class <code>istreambuf_iterator</code>	531
24.5.3.1	Template class <code>istreambuf_iterator::proxy</code>	532
24.5.3.2	<code>istreambuf_iterator</code> constructors	533
24.5.3.3	<code>istreambuf_iterator::operator*</code>	533
24.5.3.4	<code>istreambuf_iterator::operator++</code>	533
24.5.3.5	<code>istreambuf_iterator::equal</code>	533
24.5.3.6	<code>operator==</code>	533
24.5.3.7	<code>operator!=</code>	534
24.5.4	Template class <code>ostreambuf_iterator</code>	534
24.5.4.1	<code>ostreambuf_iterator</code> constructors	534
24.5.4.2	<code>ostreambuf_iterator</code> operations	534
25	Algorithms library	537
25.1	Non-modifying sequence operations	545
25.1.1	For each	545
25.1.2	Find	546
25.1.3	Find End	546
25.1.4	Find First	546
25.1.5	Adjacent find	547
25.1.6	Count	547
25.1.7	Mismatch	547
25.1.8	Equal	548
25.1.9	Search	548
25.2	Mutating sequence operations	549
25.2.1	Copy	549
25.2.2	Swap	549
25.2.3	Transform	550
25.2.4	Replace	550
25.2.5	Fill	551
25.2.6	Generate	551
25.2.7	Remove	551
25.2.8	Unique	552
25.2.9	Reverse	552
25.2.10	Rotate	553
25.2.11	Random shuffle	553
25.2.12	Partitions	554
25.3	Sorting and related operations	554
25.3.1	Sorting	555
25.3.1.1	<code>sort</code>	555
25.3.1.2	<code>stable_sort</code>	555
25.3.1.3	<code>partial_sort</code>	555
25.3.1.4	<code>partial_sort_copy</code>	556
25.3.2	Nth element	556
25.3.3	Binary search	556

25.3.3.1	lower_bound.....	556
25.3.3.2	upper_bound.....	557
25.3.3.3	equal_range.....	557
25.3.3.4	binary_search.....	557
25.3.4	Merge.....	558
25.3.5	Set operations on sorted structures	558
25.3.5.1	includes	559
25.3.5.2	set_union.....	559
25.3.5.3	set_intersection.....	559
25.3.5.4	set_difference.....	560
25.3.5.5	set_symmetric_difference.....	560
25.3.6	Heap operations	560
25.3.6.1	push_heap.....	561
25.3.6.2	pop_heap.....	561
25.3.6.3	make_heap.....	561
25.3.6.4	sort_heap.....	561
25.3.7	Minimum and maximum	562
25.3.8	Lexicographical comparison.....	562
25.3.9	Permutation generators	563
25.4	C library algorithms.....	563
26	Numerics library	565
26.1	Numeric type requirements.....	565
26.2	Complex numbers.....	566
26.2.1	Header <complex> synopsis	566
26.2.2	Template class complex.....	567
26.2.3	complex specializations	569
26.2.4	complex member functions.....	570
26.2.5	complex member operators.....	570
26.2.6	complex non-member operations	571
26.2.7	complex value operations	572
26.2.8	complex transcendentals.....	573
26.3	Numeric arrays.....	574
26.3.1	Header <valarray> synopsis	574
26.3.2	Template class valarray	577
26.3.2.1	valarray constructors.....	578
26.3.2.2	valarray assignment	579
26.3.2.3	valarray element access	580
26.3.2.4	valarray subset operations	580
26.3.2.5	valarray unary operators	580
26.3.2.6	valarray computed assignment	581
26.3.2.7	valarray member functions	581
26.3.3	valarray non-member operations	583
26.3.3.1	valarray binary operators	583
26.3.3.2	valarray logical operators	584
26.3.3.3	valarray transcendentals.....	585
26.3.4	Class slice.....	585
26.3.4.1	slice constructors.....	585
26.3.4.2	slice access functions	586
26.3.5	Template class slice_array.....	586

26.3.5.1	slice_array constructors	587
26.3.5.2	slice_array assignment	587
26.3.5.3	slice_array computed assignment	587
26.3.5.4	slice_array fill function	587
26.3.6	The gslice class	587
26.3.6.1	gslice constructors	588
26.3.6.2	gslice access functions	589
26.3.7	Template class gslice_array	589
26.3.7.1	gslice_array constructors	589
26.3.7.2	gslice_array assignment	590
26.3.7.3	gslice_array computed assignment	590
26.3.7.4	gslice_array fill function	590
26.3.8	Template class mask_array	590
26.3.8.1	mask_array constructors	591
26.3.8.2	mask_array assignment	591
26.3.8.3	mask_array computed assignment	591
26.3.8.4	mask_array fill function	592
26.3.9	Template class indirect_array	592
26.3.9.1	indirect_array constructors	592
26.3.9.2	indirect_array assignment	593
26.3.9.3	indirect_array computed assignment	593
26.3.9.4	indirect_array fill function	593
26.4	Generalized numeric operations	593
26.4.1	Accumulate	594
26.4.2	Inner product	595
26.4.3	Partial sum	595
26.4.4	Adjacent difference	595
26.5	C Library	596
27	Input/output library	599
27.1	Iostreams requirements	599
27.1.1	Imbue Limitations	599
27.1.2	Positioning Type Limitations	599
27.2	Forward declarations	599
27.3	Standard istream objects	602
27.3.1	Narrow stream objects	602
27.3.2	Wide stream objects	603
27.4	Iostreams base classes	604
27.4.1	Types	604
27.4.2	Class ios_base	605
27.4.2.1	Types	607
27.4.2.1.1	Class ios_base::failure	607
27.4.2.1.2	Type ios_base::fmtflags	607
27.4.2.1.3	Type ios_base::iostate	608
27.4.2.1.4	Type ios_base::openmode	609
27.4.2.1.5	Type ios_base::seekdir	609
27.4.2.1.6	Class ios_base::Init	609
27.4.2.2	ios_base fmtflags state functions	610

27.4.2.3	ios_base locale functions	611
27.4.2.4	ios_base static members	611
27.4.2.5	ios_base storage functions	611
27.4.2.6	ios_base callbacks	612
27.4.2.7	ios_base constructors/destructors	612
27.4.3	Template class fpos	612
27.4.3.1	fpos Members	612
27.4.3.2	fpos requirements	612
27.4.4	Template class basic_ios	613
27.4.4.1	basic_ios constructors	614
27.4.4.2	Member functions	615
27.4.4.3	basic_ios iostate flags functions	616
27.4.5	ios_base manipulators	617
27.4.5.1	fmtflags manipulators	617
27.4.5.2	adjustfield manipulators	618
27.4.5.3	basefield manipulators	619
27.4.5.4	floatfield manipulators	619
27.5	Stream buffers	619
27.5.1	Stream buffer requirements	620
27.5.2	Template class basic_streambuf<charT, traits>	620
27.5.2.1	basic_streambuf constructors	622
27.5.2.2	basic_streambuf public member functions	623
27.5.2.2.1	Locales	623
27.5.2.2.2	Buffer management and positioning	623
27.5.2.2.3	Get area	623
27.5.2.2.4	Putback	624
27.5.2.2.5	Put area	624
27.5.2.3	basic_streambuf protected member functions	624
27.5.2.3.1	Get area access	624
27.5.2.3.2	Put area access	625
27.5.2.4	basic_streambuf virtual functions	625
27.5.2.4.1	Locales	625
27.5.2.4.2	Buffer management and positioning	625
27.5.2.4.3	Get area	626
27.5.2.4.4	Putback	627
27.5.2.4.5	Put area	628
27.6	Formatting and manipulators	629
27.6.1	Input streams	630
27.6.1.1	Template class basic_istream	630
27.6.1.1.1	basic_istream constructors	632
27.6.1.1.2	Class basic_istream::sentry	632
27.6.1.2	Formatted input functions	633
27.6.1.2.1	Common requirements	633
27.6.1.2.2	Arithmetic Extractors	633
27.6.1.2.3	basic_istream::operator>>	634
27.6.1.3	Unformatted input functions	635
27.6.1.4	Standard basic_istream manipulators	639
27.6.1.5	Template class basic_iostream	639
27.6.1.5.1	basic_iostream constructors	640
27.6.1.5.2	basic_iostream destructor	640
27.6.2	Output streams	640
27.6.2.1	Template class basic_ostream	640

27.6.2.2	<code>basic_ostream</code> constructors.....	642
27.6.2.3	Class <code>basic_ostream::sentry</code>	642
27.6.2.4	<code>basic_ostream</code> seek members	643
27.6.2.5	Formatted output functions.....	643
27.6.2.5.1	Common requirements.....	643
27.6.2.5.2	Arithmetic Inserters	643
27.6.2.5.3	<code>basic_ostream::operator<<</code>	644
27.6.2.5.4	Character inserter template functions	645
27.6.2.6	Unformatted output functions.....	645
27.6.2.7	Standard <code>basic_ostream</code> manipulators	646
27.6.3	Standard manipulators	646
27.7	String-based streams	648
27.7.1	Template class <code>basic_stringbuf</code>	649
27.7.1.1	<code>basic_stringbuf</code> constructors.....	650
27.7.1.2	Member functions.....	650
27.7.1.3	Overridden virtual functions.....	651
27.7.2	Template class <code>basic_istream</code>	653
27.7.2.1	<code>basic_istream</code> constructors	653
27.7.2.2	Member functions.....	654
27.7.3	Class <code>basic_ostringstream</code>	654
27.7.3.1	<code>basic_ostringstream</code> constructors	655
27.7.3.2	Member functions.....	655
27.7.4	Template class <code>basic_stringstream</code>	655
27.7.5	<code>basic_stringstream</code> constructors.....	656
27.7.6	Member functions.....	656
27.8	File-based streams	657
27.8.1	File streams.....	657
27.8.1.1	Template class <code>basic_filebuf</code>	657
27.8.1.2	<code>basic_filebuf</code> constructors.....	658
27.8.1.3	Member functions.....	659
27.8.1.4	Overridden virtual functions.....	660
27.8.1.5	Template class <code>basic_ifstream</code>	662
27.8.1.6	<code>basic_ifstream</code> constructors	663
27.8.1.7	Member functions.....	663
27.8.1.8	Template class <code>basic_ofstream</code>	664
27.8.1.9	<code>basic_ofstream</code> constructors	664
27.8.1.10	Member functions.....	665
27.8.1.11	Template class <code>basic_fstream</code>	665
27.8.1.12	<code>basic_fstream</code> constructors.....	666
27.8.1.13	Member functions.....	666
27.8.2	C Library files.....	666
Annex A (informative)	Grammar summary	667
A.1	Keywords	667
A.2	Lexical conventions	667
A.3	Basic concepts.....	671
A.4	Expressions	671

A.5	Statements	674
A.6	Declarations	675
A.7	Declarators	677
A.8	Classes	679
A.9	Derived classes.....	680
A.10	Special member functions.....	680
A.11	Overloading	680
A.12	Templates.....	681
A.13	Exception handling	681
A.14	Preprocessing directives.....	682
Annex B (informative)	Implementation quantities.....	685
Annex C (informative)	Compatibility	687
C.1	C++ and ISO C	687
C.1.1	Clause 2: lexical conventions.....	687
C.1.2	Clause 3: basic concepts	688
C.1.3	Clause 5: expressions	690
C.1.4	Clause 6: statements.....	690
C.1.5	Clause 7: declarations	691
C.1.6	Clause 8: declarators	693
C.1.7	Clause 9: classes.....	694
C.1.8	Clause 12: special member functions.....	695
C.1.9	Clause 16: preprocessing directives	696
C.2	Standard C library	696
C.2.1	Modifications to headers	698
C.2.2	Modifications to definitions	698
C.2.2.1	Type <code>wchar_t</code>	698
C.2.2.2	Header <code><iso646.h></code>	699
C.2.2.3	Macro <code>NULL</code>	699
C.2.3	Modifications to declarations.....	699
C.2.4	Modifications to behavior	699
C.2.4.1	Macro <code>offsetof(<i>type</i>, <i>member-designator</i>)</code>	699
C.2.4.2	Memory allocation functions	699
Annex D (normative)	Compatibility features	701
D.1	Postfix increment operator	701
D.2	<code>static</code> keyword	701
D.3	Access declarations	701

D.4	Implicit conversion from const strings	701
D.5	Standard C library headers.....	701
D.6	Old iostreams members	701
D.7	char* streams	703
D.7.1	Class <code>strstreambuf</code>	703
D.7.1.1	<code>strstreambuf</code> constructors	705
D.7.1.2	Member functions	706
D.7.1.3	<code>strstreambuf</code> overridden virtual functions	706
D.7.2	Class <code>istream</code>	709
D.7.2.1	<code>istream</code> constructors	709
D.7.2.2	Member functions	709
D.7.3	Class <code>ostream</code>	710
D.7.3.1	<code>ostream</code> constructors	710
D.7.3.2	Member functions	710
D.7.4	Class <code>strstream</code>	711
D.7.4.1	<code>strstream</code> constructors	711
D.7.4.2	<code>strstream</code> destructor	712
D.7.4.3	<code>strstream</code> operations	712
Annex E (normative)	Universal-character-names	713
Index		715

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 14882 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Annexes D and E form an integral part of this International Standard. Annexes A to C are for information only.

Programming languages – C++

1 General

[intro]

1.1 Scope

[intro.scope]

- 1 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 9899:1990 *Programming languages – C* (1.2). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, inline functions, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

1.2 Normative references

[intro.refs]

- 1 The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.
 - ISO/IEC 2382 (all parts), *Information technology – Vocabulary*
 - ISO/IEC 9899:1990, *Programming languages – C*
 - ISO/IEC 9899/Amd.1:1995, *Programming languages – C, AMENDMENT 1: C Integrity*
 - ISO/IEC 10646-1:1993 *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*
- 2 The library described in clause 7 of ISO/IEC 9899:1990 and clause 7 of ISO/IEC 9899/Amd.1:1995 is hereinafter called the *Standard C Library*.¹⁾

1.3 Definitions

[intro.defs]

- 1 For the purposes of this International Standard, the definitions given in ISO/IEC 2382 and the following definitions apply. 17.1 defines additional terms that are used only in clauses 17 through 27.
- 2 Terms that are used only in a small portion of this International Standard are defined where they are used and italicized where they are defined.

1.3.1 argument

[defs.argument]

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

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

1.3.2 diagnostic message

1 General

1.3.2 diagnostic message**[defns.diagnostic]**

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

1.3.3 dynamic type**[defns.dynamic.type]**

the type of the most derived object (1.8) to which the lvalue denoted by an lvalue expression refers. [Example: if a pointer (8.3.1) *p* whose static type is "pointer to class B" is pointing to an object of class D, derived from B (clause 10), the dynamic type of the expression **p* is "D." References (8.3.2) are treated similarly.] The dynamic type of an rvalue expression is its static type.

1.3.4 ill-formed program**[defns.ill.formed]**

input to a C++ implementation that is not a well-formed program (1.3.14).

1.3.5 implementation-defined behavior**[defns.impl.defined]**

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

1.3.6 implementation limits**[defns.impl.limits]**

restrictions imposed upon programs by the implementation.

1.3.7 locale-specific behavior**[defns.locale.specific]**

behavior that depends on local conventions of nationality, culture, and language that each implementation shall document.

1.3.8 multibyte character**[defns.multibyte]**

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

1.3.9 parameter**[defns.parameter]**

an object or reference declared as part of a function declaration or definition, or in the catch clause of an exception handler, that acquires a value on entry to the function or handler; an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition; or a *template-parameter*. Parameters are also known as *formal arguments* or *formal parameters*.

1.3.10 signature**[defns.signature]**

the information about a function that participates in overload resolution (13.3): the types of its parameters and, if the function is a class member, the cv- qualifiers (if any) on the function itself and the class in which the member function is declared.²⁾ The signature of a template function specialization includes the types of its template arguments (14.5.5.1).

1.3.11 static type**[defns.static.type]**

the type of an expression (3.9), which type results from analysis of the program without considering execution semantics. The static type of an expression depends only on the form of the program in which the expression appears, and does not change while the program is executing.

1.3.12 undefined behavior**[defns.undefined]**

behavior, such as might arise upon use of an erroneous program construct or erroneous data, for which this International Standard imposes no requirements. Undefined behavior may also be expected when this International Standard omits the description of any explicit definition of behavior. [Note: permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). Many erroneous program constructs do not engender undefined behavior; they are

²⁾ Function signatures do not include return type, because that does not participate in overload resolution.

1 General**1.3.12 undefined behavior**

required to be diagnosed.]

1.3.13 unspecified behavior**[defns.unspecified]**

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

1.3.14 well-formed program**[defns.well.formed]**

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

1.4 Implementation compliance**[intro.compliance]**

- 1 The set of *diagnosable rules* consists of all syntactic and semantic rules in this International Standard except for those rules containing an explicit notation that “no diagnostic is required” or which are described as resulting in “undefined behavior.”
- 2 Although this International Standard states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:
 - If a program contains no violations of the rules in this International Standard, a conforming implementation shall, within its resource limits, accept and correctly execute³⁾ that program.
 - If a program contains a violation of any diagnosable rule, a conforming implementation shall issue at least one diagnostic message, except that
 - If a program contains a violation of a rule for which no diagnostic is required, this International Standard places no requirement on implementations with respect to that program.
- 3 For classes and class templates, the library clauses specify partial definitions. Private members (clause 11) are not specified, but each implementation shall supply them to complete the definitions according to the description in the library clauses.
- 4 For functions, function templates, objects, and values, the library clauses specify declarations. Implementations shall supply definitions consistent with the descriptions in the library clauses.
- 5 The names defined in the library have namespace scope (7.3). A C++ translation unit (2.1) obtains access to these names by including the appropriate standard library header (16.2).
- 6 The templates, classes, functions, and objects in the library have external linkage (3.5). The implementation provides definitions for standard library entities, as necessary, while combining translation units to form a complete C++ program (2.1).
- 7 Two kinds of implementations are defined: *hosted* and *freestanding*. For a hosted implementation, this International Standard defines the set of available libraries. A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation-defined set of libraries that includes certain language-support libraries (17.4.1.3).
- 8 A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any well-formed program. Implementations are required to diagnose programs that use such extensions that are ill-formed according to this International Standard. Having done so, however, they can compile and execute such programs.

³⁾ “Correct execution” can include undefined behavior, depending on the data being processed; see 1.3 and 1.9.

1.5 Structure of this International Standard

1 General

1.5 Structure of this International Standard

[intro.structure]

- 1 Clauses 2 through 16 describe the C++ programming language. That description includes detailed syntactic specifications in a form described in 1.6. For convenience, Annex A repeats all such syntactic specifications.
- 2 Clauses 17 through 27 (the *library clauses*) describe the Standard C++ library, which provides definitions for the following kinds of entities: macros (16.3), values (clause 3), types (8.1, 8.3), templates (clause 14), classes (clause 9), functions (8.3.5), and objects (clause 7).
- 3 Annex B recommends lower bounds on the capacity of conforming implementations.
- 4 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.
- 5 Finally, Annex E says what characters are valid in universal-character names in C++ identifiers (2.10).
- 6 Throughout this International Standard, each example is introduced by “[*Example:*” and terminated by “]”. Each note is introduced by “[*Note:*” and terminated by “]”. Examples and notes may be nested.

1.6 Syntax notation

[syntax]

- 1 In the syntax notation used in this International Standard, syntactic categories are indicated by *italic* type, and literal words and characters in **constant width** type. Alternatives are listed on separate lines except in a few cases where a long set of alternatives is presented on one line, marked by the phrase “one of.” An optional terminal or nonterminal symbol is indicated by the subscript “*opt*,” so

{ *expression_{opt}* }

indicates an optional expression enclosed in braces.

- 2 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]

- 1 The fundamental storage unit in the C++ memory model is the *byte*. A byte is at least large enough to contain any member of the basic execution character set and 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.
- 2 [*Note:* the representation of types is described in 3.9.]

1.8 The C++ object model

[intro.object]

- 1 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.] 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.

1 General**1.8 The C++ object model**

Some objects are *polymorphic* (10.3); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the *expressions* (clause 5) used to access them.

- 2 Objects can contain other objects, called *sub-objects*. A sub-object can be a *member sub-object* (9.2), a *base class sub-object* (clause 10), or an array element. An object that is not a sub-object of any other object is called a *complete object*.
- 3 For every object *x*, there is some object called *the complete object of x*, determined as follows:
 - If *x* is a complete object, then *x* is the complete object of *x*.
 - Otherwise, the complete object of *x* is the complete object of the (unique) object that contains *x*.
- 4 If a complete object, a data member (9.2), or an array element is of class type, its type is considered the *most derived* class, to distinguish it from the class type of any base class subobject; an object of a most derived class type is called a *most derived object*.
- 5 Unless it is a bit-field (9.6), a most derived object shall have a non-zero size and shall occupy one or more bytes of storage. Base class sub-objects may have zero size. An object of POD⁴⁾ type (3.9) shall occupy contiguous bytes of storage.
- 6 [Note: C++ provides a variety of built-in types and several ways of composing new types from existing types (3.9).]

1.9 Program execution**[intro.execution]**

- 1 The semantic descriptions in this International Standard define a parameterized nondeterministic abstract machine. This International Standard places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.⁵⁾
- 2 Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, `sizeof(int)`). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects. Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the “corresponding instance” below).
- 3 Certain other aspects and operations of the abstract machine are described in this International Standard as unspecified (for example, order of evaluation of arguments to a function). Where possible, this International Standard defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution sequence for a given program and a given input.
- 4 Certain other operations are described in this International Standard as undefined (for example, the effect of dereferencing the null pointer). [Note: this International Standard imposes no requirements on the behavior of programs that contain undefined behavior.]
- 5 A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible execution sequences of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution sequence contains an undefined operation, this International Standard places no requirement on the implementation executing that program

⁴⁾ The acronym POD stands for “plain old data.”

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

1.9 Program execution

1 General

with that input (not even with regard to operations preceding the first undefined operation).

- 6 The observable behavior of the abstract machine is its sequence of reads and writes to `volatile` data and calls to library I/O functions.⁶⁾
- 7 Accessing an object designated by a `volatile` lvalue (3.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. Evaluation of an expression might produce side effects. At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.⁷⁾
- 8 Once the execution of a function begins, no expressions from the calling function are evaluated until execution of the called function has completed.⁸⁾
- 9 When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects with type other than `volatile sig_atomic_t` are unspecified, and the value of any object not of `volatile sig_atomic_t` that is modified by the handler becomes undefined.
- 10 An instance of each object with automatic storage duration (3.7.2) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).
- 11 The least requirements on a conforming implementation are:
- At sequence points, volatile objects are stable in the sense that previous evaluations are complete and subsequent evaluations have not yet occurred.
 - At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.
 - The input and output dynamics of interactive devices shall take place in such a fashion that prompting messages actually appear prior to a program waiting for input. What constitutes an interactive device is implementation-defined.

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

- 12 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.
- 13 [Note: certain contexts in C++ cause the evaluation of a full-expression that results from a syntactic construct other than *expression* (5.18). For example, in 8.5 one syntax for *initializer* is

(*expression-list*)

but the resulting construct is a function call upon a constructor function with *expression-list* as an argument list; such a function call is a full-expression. For example, in 8.5, another syntax for *initializer* is

= *initializer-clause*

but again the resulting construct might be a function call upon a constructor function with one *assignment-expression* as an argument; again, the function call is a full-expression.]

⁶⁾ An implementation can offer additional library I/O functions as an extension. Implementations that do so should treat calls to those functions as “observable behavior” as well.

⁷⁾ Note that some aspects of sequencing in the abstract machine are unspecified; the preceding restriction upon side effects applies to that particular execution sequence in which the actual code is generated. Also note that when a call to a library I/O function returns, the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) may not have completed yet.

⁸⁾ In other words, function executions do not interleave with each other.

1 General

1.9 Program execution

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

15 [Note: operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative.⁹⁾ 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));
```

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.]

16 There is a sequence point at the completion of evaluation of each full-expression¹⁰⁾.

17 When calling a function (whether or not the function is inline), there is a sequence point after the evaluation of all function arguments (if any) which takes place before execution of any expressions or statements in the function body. There is also a sequence point after the copying of a returned value and before the execution of any expressions outside the function¹¹⁾. Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. [Example: evaluation of a `new` expression invokes one or more allocation and constructor functions; see 5.3.4. For another example, invocation of a conversion function (12.3.2) can arise in contexts in which no function call syntax appears.] The sequence points at function-entry and function-exit (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

18 In the evaluation of each of the expressions

```
a && b
a || b
a ? b : c
a , b
```

using the built-in meaning of the operators in these expressions (5.14, 5.15, 5.16, 5.18), there is a sequence

⁹⁾ Overloaded operators are never assumed to be associative or commutative.

¹⁰⁾ As specified in 12.2, after the "end-of-full-expression" sequence point, 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.

¹¹⁾ The sequence point at the function return is not explicitly specified in ISO C, and can be considered redundant with sequence points at full-expressions, but the extra clarity is important in C++. In C++, there are more ways in which a called function can terminate its execution, such as the throw of an exception.

1.9 Program execution**1 General**

point after the evaluation of the first expression¹²⁾.

1.10 Acknowledgments**[intro.ack]**

- 1 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).
- 2 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).
- 3 All rights in these originals are reserved.

¹²⁾ The operators indicated in this paragraph are the built-in operators, as described in clause 5. When one of these operators is overloaded (clause 13) in a valid context, thus designating a user-defined operator function, the expression designates a function invocation, and the operands form an argument list, without an implied sequence point between them.

2 Lexical conventions

[lex]

- 1 The text of the program is kept in units called *source files* in this International Standard. A source file together with all the headers (17.4.1.2) and source files included (16.2) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (16.1) preprocessing directives, is called a *translation unit*. [Note: a C++ program need not all be translated at the same time.]
- 2 [Note: previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (3.5) by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program. (3.5).]

2.1 Phases of translation

[lex.phases]

- 1 The precedence among the syntax rules of translation is specified by the following phases.¹³⁾
 - 1 Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences (2.3) are replaced by corresponding single-character internal representations. Any source file character not in the basic source character set (2.2) is replaced by the universal-character-name that designates that character. (An implementation may use any internal encoding, so long as an actual extended character encountered in the source file, and the same extended character expressed in the source file as a universal-character-name (i.e. using the `\uXXXX` notation), are handled equivalently.)
 - 2 Each instance of a new-line character and an immediately preceding backslash character is deleted, splicing physical source lines to form logical source lines. If, as a result, a character sequence that matches the syntax of a universal-character-name is produced, the behavior is undefined. If a source file that is not empty does not end in a new-line character, or ends in a new-line character immediately preceded by a backslash character, the behavior is undefined.
 - 3 The source file is decomposed into preprocessing tokens (2.4) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or partial comment¹⁴⁾. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined. 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.]
 - 4 Preprocessing directives are executed and macro invocations are expanded. If a character sequence that matches the syntax of a universal-character-name is produced by token concatenation (16.3.3), the behavior is undefined. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.
 - 5 Each source character set member, escape sequence, or universal-character-name in character literals and string literals is converted to a member of the execution character set (2.13.2, 2.13.4).
 - 6 Adjacent ordinary string literal tokens are concatenated. Adjacent wide string literal tokens are concatenated.
 - 7 White-space characters separating tokens are no longer significant. Each preprocessing token is

¹³⁾ Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.

¹⁴⁾ 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.

2.1 Phases of translation

2 Lexical conventions

converted into a token. (2.6). The resulting tokens are syntactically and semantically analyzed and translated. [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.]

- 8 Translated translation units and instantiation units are combined as follows: [Note: some or all of these may be supplied from a library.] 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).] 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.] 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.] The program is ill-formed if any instantiation fails.
- 9 All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

2.2 Character sets

[lex.charset]

- 1 The *basic source character set* consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:¹⁵⁾

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

```

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

hex-quad:

hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

universal-character-name:

\u hex-quad

\U hex-quad hex-quad

The character designated by the universal-character-name `\UNNNNNNNN` is that character whose character short name in ISO/IEC 10646 is `NNNNNNNN`; the character designated by the universal-character-name `\UNNNN` is that character whose character short name in ISO/IEC 10646 is `0000NNNN`. If the hexadecimal value for a universal character name is less than 0x20 or in the range 0x7F-0x9F (inclusive), or if the universal character name designates a character in the basic source character set, then the program is ill-formed.

- 3 The *basic execution character set* and the *basic execution wide-character set* shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a *null character* (respectively, *null wide character*), whose representation has all zero bits. For each basic execution character set, the values of the members shall be non-negative and distinct from one another. The *execution character set* and the *execution wide-character set* are supersets of the basic execution character set and the basic execution wide-character set, respectively. The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.

¹⁵⁾ 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.

2 Lexical conventions

2.3 Trigraph sequences

2.3 Trigraph sequences

[lex.trigraph]

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

Table 1—trigraph sequences

<i>trigraph</i>	<i>replacement</i>	<i>trigraph</i>	<i>replacement</i>	<i>trigraph</i>	<i>replacement</i>
??=	#	??(	[	??<	{
??/	\	??)	]	??>	}
??'	^	??!		??-	~

- 2 [Example:

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

becomes

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

—end example]

- 3 No other trigraph sequence exists. Each ? that does not begin one of the trigraphs listed above is not changed.

2.4 Preprocessing tokens

[lex.pptoken]

preprocessing-token:

header-name

identifier

pp-number

character-literal

string-literal

preprocessing-op-or-punc

each non-white-space character that cannot be one of the above

- 1 Each preprocessing token that is converted to a token (2.6) shall have the lexical form of a keyword, an identifier, a literal, an operator, or a punctuation.
- 2 A *preprocessing token* is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: *header names*, *identifiers*, *preprocessing numbers*, *character literals*, *string literals*, *preprocessing-op-or-punc*, and single non-white-space characters that do not lexically match the other preprocessing token categories. If a ' or a " character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by *white space*; this consists of comments (2.7), or *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in clause 16, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.
- 3 If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail.
- 4 [Example: The program fragment 1Ex is parsed as a preprocessing number token (one that is not a valid floating or integer literal token), even though a parse as the pair of preprocessing tokens 1 and EX might produce a valid expression (for example, if EX were a macro defined as +1). Similarly, the program fragment 1E1 is parsed as a preprocessing number (one that is a valid floating literal token), whether or not E is a macro name.]

2.4 Preprocessing tokens

2 Lexical conventions

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

2.5 Alternative tokens

[lex.digraph]

- 1 Alternative token representations are provided for some operators and punctuators¹⁶⁾.
- 2 In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling¹⁷⁾. The set of alternative tokens is defined in Table 2.

Table 2—alternative tokens

<i>alternative</i>	<i>primary</i>	<i>alternative</i>	<i>primary</i>	<i>alternative</i>	<i>primary</i>
<%	{	and	&&	and_eq	&=
%>	}	bitor		or_eq	=
<:	[	or		xor_eq	^=
:>	]	xor	^	not	!
%:	#	compl	~	not_eq	!=
%::	##	bitand	&		

2.6 Tokens

[lex.token]

token:

identifier
keyword
literal
operator
punctuator

- 1 There are five kinds of tokens: identifiers, keywords, literals,¹⁸⁾ operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, “white space”), as described below, are ignored except as they serve to separate tokens. [Note: Some white space is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters.]

2.7 Comments

[lex.comment]

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

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

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

¹⁸⁾ Literals include strings and character and numeric literals.

2 Lexical conventions

2.8 Header names

2.8 Header names

[lex.header]

header-name:

<h-char-sequence>
"q-char-sequence"

h-char-sequence:

h-char
h-char-sequence h-char

h-char:

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

q-char-sequence:

q-char
q-char-sequence q-char

q-char:

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

- 1 Header name preprocessing tokens shall only appear within a `#include` preprocessing directive (16.2). The sequences in both forms of *header-names* are mapped in an implementation-defined manner to headers or to external source file names as specified in 16.2.
- 2 If either of the characters ' or \, or either of the character sequences /* or // appears in a *q-char-sequence* or a *h-char-sequence*, or the character " appears in a *h-char-sequence*, the behavior is undefined.¹⁹⁾

2.9 Preprocessing numbers

[lex.pppnumber]

pp-number:

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

- 1 Preprocessing number tokens lexically include all integral literal tokens (2.13.1) and all floating literal tokens (2.13.3).
- 2 A preprocessing number does not have a type or a value; it acquires both after a successful conversion (as part of translation phase 7, 2.1) to an integral literal token or a floating literal token.

2.10 Identifiers

[lex.name]

identifier:

nondigit
identifier nondigit
identifier digit

¹⁹⁾ Thus, sequences of characters that resemble escape sequences cause undefined behavior.

2.10 Identifiers

2 Lexical conventions

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

digit: one of
 0 1 2 3 4 5 6 7 8 9

- 1 An identifier is an arbitrarily long sequence of letters and digits. Each universal-character-name in an identifier shall designate a character whose encoding in ISO 10646 falls into one of the ranges specified in Annex E. Upper- and lower-case letters are different. All characters are significant.²⁰⁾
- 2 In addition, some identifiers are reserved for use by C++ implementations and standard libraries (17.4.3.1.2) and shall not be used otherwise; no diagnostic is required.

2.11 Keywords

[lex.key]

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

Table 3—keywords

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

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

Table 4—alternative representations

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	xor	xor_eq	

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

2 Lexical conventions

2.12 Operators and punctuators

2.12 Operators and punctuators

[lex.operators]

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

preprocessing-op-or-punc: one of

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

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

2.13 Literals

[lex.literal]

- 1 There are several kinds of literals.²¹⁾

literal:

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

2.13.1 Integer literals

[lex.icon]

integer-literal:

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

decimal-literal:

nonzero-digit
decimal-literal digit

octal-literal:

0
octal-literal octal-digit

hexadecimal-literal:

0x *hexadecimal-digit*
 0X *hexadecimal-digit*
hexadecimal-literal hexadecimal-digit

nonzero-digit: one of

1 2 3 4 5 6 7 8 9

octal-digit: one of

0 1 2 3 4 5 6 7

²¹⁾ The term “literal” generally designates, in this International Standard, those tokens that are called “constants” in ISO C.

2.13.1 Integer literals

2 Lexical conventions

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

integer-suffix:
unsigned-suffix long-suffix_{opt}
long-suffix unsigned-suffix_{opt}

unsigned-suffix: one of
 u U

long-suffix: one of
 l L

- 1 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.²²⁾ A *hexadecimal* integer literal (base sixteen) begins with 0x or 0X and consists of a sequence of hexadecimal digits, which include the decimal digits and the letters a through f and A through F with decimal values ten through fifteen. [Example: the number twelve can be written 12, 014, or 0XC.]
- 2 The type of an integer literal depends on its form, value, and suffix. If it is decimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `long int`; if the value cannot be represented as a `long int`, the behavior is undefined. If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `unsigned int`, `long int`, `unsigned long int`. If it is suffixed by u or U, its type is the first of these types in which its value can be represented: `unsigned int`, `unsigned long int`. If it is suffixed by l or L, its type is the first of these types in which its value can be represented: `long int`, `unsigned long int`. If it is suffixed by ul, lu, uL, Lu, UL, or LU, its type is `unsigned long int`.
- 3 A program is ill-formed if one of its translation units contains an integer literal that cannot be represented by any of the allowed types.

2.13.2 Character literals

[lex.ccon]

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

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

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

²²⁾ The digits 8 and 9 are not octal digits.

2 Lexical conventions

2.13.2 Character literals

escape-sequence:

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

simple-escape-sequence: one of

`\'` `\"` `\?` `\\`
`\a` `\b` `\f` `\n` `\r` `\t` `\v`

octal-escape-sequence:

`\` *octal-digit*
`\` *octal-digit* *octal-digit*
`\` *octal-digit* *octal-digit* *octal-digit*

hexadecimal-escape-sequence:

`\x` *hexadecimal-digit*
hexadecimal-escape-sequence *hexadecimal-digit*

- 1 A character literal is one or more characters enclosed in single quotes, as in `'x'`, optionally preceded by the letter `L`, as in `L'x'`. A character literal that does not begin with `L` is an ordinary character literal, also referred to as a narrow-character literal. An ordinary character literal that contains a single *c-char* has type `char`, with value equal to the numerical value of the encoding of the *c-char* in the execution character set. An ordinary character literal that contains more than one *c-char* is a *multicharacter literal*. A multicharacter literal has type `int` and implementation-defined value.
- 2 A character literal that begins with the letter `L`, such as `L'x'`, is a wide-character literal. A wide-character literal has type `wchar_t`.²³⁾ The value of a wide-character literal containing a single *c-char* has value equal to the numerical value of the encoding of the *c-char* in the execution wide-character set. The value of a wide-character literal containing multiple *c-chars* is implementation-defined.
- 3 Certain nongraphic characters, the single quote `'`, the double quote `"`, the question mark `?`, and the backslash `\`, can be represented according to Table 5.

Table 5—escape sequences

new-line	NL (LF)	<code>\n</code>
horizontal tab	HT	<code>\t</code>
vertical tab	VT	<code>\v</code>
backspace	BS	<code>\b</code>
carriage return	CR	<code>\r</code>
form feed	FF	<code>\f</code>
alert	BEL	<code>\a</code>
backslash	<code>\</code>	<code>\\</code>
question mark	<code>?</code>	<code>\?</code>
single quote	<code>'</code>	<code>\'</code>
double quote	<code>"</code>	<code>\"</code>
octal number	<i>ooo</i>	<code>\ooo</code>
hex number	<i>hhh</i>	<code>\xhhh</code>

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. If the character following a backslash is not one of those specified, the behavior is undefined. An escape sequence specifies a single character.

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

2.13.2 Character literals

2 Lexical conventions

- 4 The escape `\ooo` consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape `\xhhh` consists of the backslash followed by x followed by one or more hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of digits in a hexadecimal sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character literal is implementation-defined if it falls outside of the implementation-defined range defined for `char` (for ordinary literals) or `wchar_t` (for wide literals).
- 5 A universal-character-name is translated to the encoding, in the execution character set, of the character named. If there is no such encoding, the universal-character-name is translated to an implementation-defined encoding. [Note: in translation phase 1, a universal-character-name is introduced whenever an actual extended character is encountered in the source text. Therefore, all extended characters are described in terms of universal-character-names. However, the actual compiler implementation may use its own native character set, so long as the same results are obtained.]

2.13.3 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*

sign: one of

+ -

digit-sequence:

digit
digit-sequence *digit*

floating-suffix: one of

f l F L

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

2 Lexical conventions

2.13.3 Floating literals

2.13.4 String literals

[lex.string]

string-literal:

"*s-char-sequence*_{opt}"
 L "*s-char-sequence*_{opt}"

s-char-sequence:

s-char
s-char-sequence *s-char*

s-char:

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

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

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

- 3 In translation phase 6 (2.1), adjacent narrow string literals are concatenated and adjacent wide string literals are concatenated. If a narrow string literal token is adjacent to a wide string literal token, the behavior is undefined. 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').]

- 4 After any necessary concatenation, in translation phase 7 (2.1), '\0' is appended to every string literal so that programs that scan a string can find its end.
- 5 Escape sequences and universal-character-names in string literals have the same meaning as in character literals (2.13.2), except that the single quote ' is representable either by itself or by the escape sequence '\', and the double quote " shall be preceded by a \. In a narrow string literal, a universal-character-name may map to more than one **char** element due to *multibyte encoding*. The size of a wide string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for the terminating L'\0'. The size of a narrow string literal is the total number of escape sequences and other characters, plus at least one for the multibyte encoding of each universal-character-name, plus one for the terminating '\0'.

2.13.5 Boolean literals

[lex.bool]

boolean-literal:

false
 true

- 1 The Boolean literals are the keywords **false** and **true**. Such literals have type **bool**. They are not lvalues.

(Blank page)

3 Basic concepts

[basic]

- 1 [Note: this clause presents the basic concepts of the C++ language. It explains the difference between an *object* and a *name* and how they relate to the notion of an *lvalue*. It introduces the concepts of a *declaration* and a *definition* and presents C++'s notion of *type*, *scope*, *linkage*, and *storage duration*. The mechanisms for starting and terminating a program are discussed. Finally, this clause presents the fundamental types of the language and lists the ways of constructing *compound* types from these.
- 2 This clause does not cover concepts that affect only a single part of the language. Such concepts are discussed in the relevant clauses.]
- 3 An *entity* is a value, object, subobject, base class subobject, array element, variable, function, instance of a function, enumerator, type, class member, template, or namespace.
- 4 A *name* is a use of an identifier (2.10) that denotes an entity or *label* (6.6.4, 6.1). A *variable* is introduced by the declaration of an object. The variable's name denotes the object.
- 5 Every name that denotes an entity is introduced by a *declaration*. Every name that denotes a label is introduced either by a *goto* statement (6.6.4) or a *labeled-statement* (6.1).
- 6 Some names denote types, classes, enumerations, or templates. In general, it is necessary to determine whether or not a name denotes one of these entities before parsing the program that contains it. The process that determines this is called *name lookup* (3.4).
- 7 Two names are *the same* if
 - they are identifiers composed of the same character sequence; or
 - they are the names of overloaded operator functions formed with the same operator; or
 - they are the names of user-defined conversion functions formed with the same type.
- 8 An identifier used in more than one translation unit can potentially refer to the same entity in these translation units depending on the linkage (3.5) of the identifier specified in each translation unit.

3.1 Declarations and definitions

[basic.def]

- 1 A declaration (clause 7) introduces names into a translation unit or redeclares names introduced by previous declarations. A declaration specifies the interpretation and attributes of these names.
- 2 A declaration is a *definition* unless it declares a function without specifying the function's body (8.4), it contains the *extern* specifier (7.1.1) or a *linkage-specification*²⁴⁾ (7.5) and neither an *initializer* nor a *function-body*, it declares a static data member in a class declaration (9.4), it is a class name declaration (9.1), or it is a *typedef* declaration (7.1.3), a *using-declaration* (7.3.3), or a *using-directive* (7.3.4).

²⁴⁾ Appearing inside the braced-enclosed *declaration-seq* in a *linkage-specification* does not affect whether a declaration is a definition.

3.1 Declarations and definitions

3 Basic concepts

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

```

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

```

whereas these are just declarations:

```

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

```

—end example]

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

```

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

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

```

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

```

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

```

—end example] —end note]

- 5 [Note: a class name can also be implicitly declared by an *elaborated-type-specifier* (3.3.1).]
- 6 A program is ill-formed if the definition of any object gives the object an incomplete type (3.9).

3.2 One definition rule

[basic.def.odr]

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

3 Basic concepts

3.2 One definition rule

- 2 An expression is *potentially evaluated* unless either it is the operand of the `sizeof` operator (5.3.3), or it is the operand of the `typeid` operator and does not designate an lvalue of polymorphic class type (5.2.8). An object or non-overloaded function is *used* if its name appears in a potentially-evaluated expression. A virtual member function is used if it is not pure. An overloaded function is used if it is selected by overload resolution when referred to from a potentially-evaluated expression. [Note: this covers calls to named functions (5.2.2), operator overloading (clause 13), user-defined conversions (12.3.2), allocation function for placement new (5.3.4), as well as non-default initialization (8.5). A copy constructor is used even if the call is actually elided by the implementation.] An allocation or deallocation function for a class is used by a new expression appearing in a potentially-evaluated expression as specified in 5.3.4 and 12.5. A deallocation function for a class is used by a delete expression appearing in a potentially-evaluated expression as specified in 5.3.5 and 12.5. A copy-assignment function for a class is used by an implicitly-defined copy-assignment function for another class as specified in 12.8. A default constructor for a class is used by default initialization as specified in 8.5. A constructor for a class is used as specified in 8.5. A destructor for a class is used as specified in 12.4.
- 3 Every program shall contain exactly one definition of every non-inline function or object that is used in that program; no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 12.1, 12.4 and 12.8). An inline function shall be defined in every translation unit in which it is used.
- 4 Exactly one definition of a class is required in a translation unit if the class is used in a way that requires the class type to be complete. [Example: the following complete translation unit is well-formed, even though it never defines X:

```

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

```

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

- an object of type T is defined (3.1, 5.3.4), or
- an lvalue-to-rvalue conversion is applied to an lvalue referring to an object of type T (4.1), or
- an expression is converted (either implicitly or explicitly) to type T (clause 4, 5.2.3, 5.2.7, 5.2.9, 5.4), or
- an expression that is not a null pointer constant, and has type other than `void *`, is converted to the type pointer to T or reference to T using an implicit conversion (clause 4), a `dynamic_cast` (5.2.7) or a `static_cast` (5.2.9), or
- a class member access operator is applied to an expression of type T (5.2.5), or
- the `typeid` operator (5.2.8) or the `sizeof` operator (5.3.3) is applied to an operand of type T, or
- a function with a return type or argument type of type T is defined (3.1) or called (5.2.2), or
- an lvalue of type T is assigned to (5.17).]

- 5 There can be more than one definition of a class type (clause 9), enumeration type (7.2), inline function with external linkage (7.1.2), class template (clause 14), non-static function template (14.5.5), static data member of a class template (14.5.1.3), member function template (14.5.1.1), or template specialization for which some template parameters are not specified (14.7, 14.5.4) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements. Given such an entity named D defined in more than one translation unit, then
- each definition of D shall consist of the same sequence of tokens; and
 - in each definition of D, corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of D, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a `const` object with internal or no linkage if the object has the same integral or enumeration type in all definitions of D,

3.2 One definition rule

3 Basic concepts

and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of *D*; and

- in each definition of *D*, the overloaded operators referred to, the implicit calls to conversion functions, constructors, operator new functions and operator delete functions, shall refer to the same function, or to a function defined within the definition of *D*; and
- in each definition of *D*, a default argument used by an (implicit or explicit) function call is treated as if its token sequence were present in the definition of *D*; that is, the default argument is subject to the three requirements described above (and, if the default argument has sub-expressions with default arguments, this requirement applies recursively).²⁵⁾
- if *D* is a class with an implicitly-declared constructor (12.1), it is as if the constructor was implicitly defined in every translation unit where it is used, and the implicit definition in every translation unit shall call the same constructor for a base class or a class member of *D*. [Example:

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

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

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

3.3 Declarative regions and scopes

[basic.scope]

- 1 Every name is introduced in some portion of program text called a *declarative region*, which is the largest part of the program in which that name is *valid*, that is, in which that name may be used as an unqualified name to refer to the same entity. In general, each particular name is valid only within some possibly discontinuous 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.

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