

Contents

Preface	xvii
Versions of This Book	xvii
Acknowledgments	xix
About This Book	xxi
What You Should Know Before Reading This Book	xxi
Overall Structure of the Book	xxii
How to Read This Book	xxii
Error Terminology	xxii
The C++17 Standard	xxiii
Example Code and Additional Information	xxiii
Feedback	xxiii
Part I: Basic Language Features	1
1 Structured Bindings	3
1.1 Structured Bindings in Detail	4
1.2 Where Structured Bindings Can Be Used	7
1.2.1 Structures and Classes	8
1.2.2 Raw Arrays	9
1.2.3 <code>std::pair</code> , <code>std::tuple</code> , and <code>std::array</code>	9
1.3 Providing a Tuple-Like API for Structured Bindings	11
1.4 Afternotes	19
2 <code>if</code> and <code>switch</code> with Initialization	21
2.1 <code>if</code> with Initialization	21

2.2	switch with Initialization	23
2.3	Afternotes	24
3	Inline Variables	25
3.1	Motivation for Inline Variables	25
3.2	Using Inline Variables	27
3.3	constexpr Now Implies inline For Static Members	28
3.4	Inline Variables and thread_local	29
3.5	Afternotes	31
4	Aggregate Extensions	33
4.1	Motivation for Extended Aggregate Initialization	34
4.2	Using Extended Aggregate Initialization	34
4.3	Definition of Aggregates	36
4.4	Backward Incompatibilities	36
4.5	Afternotes	37
5	Mandatory Copy Elision or Passing Unmaterialized Objects	39
5.1	Motivation for Mandatory Copy Elision for Temporaries	39
5.2	Benefit of Mandatory Copy Elision for Temporaries	41
5.3	Clarified Value Categories	42
5.3.1	Value Categories	42
5.3.2	Value Categories Since C++17	44
5.4	Unmaterialized Return Value Passing	45
5.5	Afternotes	46
6	Lambda Extensions	47
6.1	constexpr Lambdas	47
6.1.1	Using constexpr Lambdas	49
6.2	Passing Copies of this to Lambdas	50
6.3	Capturing by const Reference	53
6.4	Afternotes	53
7	New Attributes and Attribute Features	55
7.1	Attribute [[nodiscard]]	55

7.2	Attribute <code>[[maybe_unused]]</code>	57
7.3	Attribute <code>[[fallthrough]]</code>	58
7.4	General Attribute Extensions	58
7.5	Afternotes	59
8	Other Language Features	61
8.1	Nested Namespaces	61
8.2	Defined Expression Evaluation Order	62
8.3	Relaxed Enum Initialization from Integral Values	65
8.4	Fixed Direct List Initialization with <code>auto</code>	66
8.5	Hexadecimal Floating-Point Literals	67
8.6	UTF-8 Character Literals	68
8.7	Exception Specifications as Part of the Type	69
8.8	Single-Argument <code>static_assert</code>	72
8.9	Preprocessor Condition <code>__has_include</code>	73
8.10	Afternotes	73
	Part II: Template Features	75
9	Class Template Argument Deduction	77
9.1	Use of Class Template Argument Deduction	77
9.1.1	Copying by Default	79
9.1.2	Deducing the Type of Lambdas	80
9.1.3	No Partial Class Template Argument Deduction	81
9.1.4	Class Template Argument Deduction Instead of Convenience Functions	83
9.2	Deduction Guides	84
9.2.1	Using Deduction Guides to Force Decay	85
9.2.2	Non-Template Deduction Guides	86
9.2.3	Deduction Guides versus Constructors	86
9.2.4	Explicit Deduction Guides	87
9.2.5	Deduction Guides for Aggregates	88
9.2.6	Standard Deduction Guides	89
9.3	Afternotes	93

10 Compile-Time <code>if</code>	95
10.1 Motivation for Compile-Time <code>if</code>	96
10.2 Using Compile-Time <code>if</code>	98
10.2.1 Caveats for Compile-Time <code>if</code>	98
10.2.2 Other Compile-Time <code>if</code> Examples	101
10.3 Compile-Time <code>if</code> with Initialization	103
10.4 Using Compile-Time <code>if</code> Outside Templates	104
10.5 Afternotes	105
11 Fold Expressions	107
11.1 Motivation for Fold Expressions	108
11.2 Using Fold Expressions	108
11.2.1 Dealing with Empty Parameter Packs	109
11.2.2 Supported Operators	112
11.2.3 Using Fold Expressions for Types	117
11.3 Afternotes	118
12 Dealing with String Literals as Template Parameters	119
12.1 Using Strings in Templates	119
12.2 Afternotes	120
13 Placeholder Types like <code>auto</code> as Template Parameters	121
13.1 Using <code>auto</code> for Template Parameters	121
13.1.1 Parameterizing Templates for Characters and Strings	122
13.1.2 Defining Metaprogramming Constants	123
13.2 Using <code>auto</code> as Variable Template Parameter	124
13.3 Using <code>decltype(auto)</code> as Template Parameter	126
13.4 Afternotes	127
14 Extended Using Declarations	129
14.1 Using Variadic Using Declarations	129
14.2 Variadic Using Declarations for Inheriting Constructors	130
14.3 Afternotes	132

Part III: New Library Components	133
15 <code>std::optional<></code>	135
15.1 Using <code>std::optional<></code>	135
15.1.1 Optional Return Values	135
15.1.2 Optional Arguments and Data Members	137
15.2 <code>std::optional<></code> Types and Operations	139
15.2.1 <code>std::optional<></code> Types	139
15.2.2 <code>std::optional<></code> Operations	139
15.3 Special Cases	146
15.3.1 Optional of Boolean or Raw Pointer Values	146
15.3.2 Optional of Optional	146
15.4 Afternotes	147
16 <code>std::variant<></code>	149
16.1 Motivation for <code>std::variant<></code>	149
16.2 Using <code>std::variant<></code>	150
16.3 <code>std::variant<></code> Types and Operations	152
16.3.1 <code>std::variant<></code> Types	152
16.3.2 <code>std::variant<></code> Operations	153
16.3.3 Visitors	157
16.3.4 Valueless by Exception	162
16.4 Polymorphism and Heterogeneous Collections with <code>std::variant</code>	162
16.4.1 Geometric Objects with <code>std::variant</code>	163
16.4.2 Other Heterogeneous Collections with <code>std::variant</code>	165
16.4.3 Comparing variant Polymorphism	167
16.5 Special Cases with <code>std::variant<></code>	168
16.5.1 Having Both <code>bool</code> and <code>std::string</code> Alternatives	168
16.6 Afternotes	169
17 <code>std::any</code>	171
17.1 Using <code>std::any</code>	171
17.2 <code>std::any</code> Types and Operations	174
17.2.1 Any Types	174
17.2.2 Any Operations	174

17.3	Afternotes	177
18	std::byte	179
18.1	Using std::byte	179
18.2	std::byte Types and Operations	181
18.2.1	std::byte Types	181
18.2.2	std::byte Operations	182
18.3	Afternotes	184
19	String Views	185
19.1	Differences Compared to std::string	185
19.2	Using String Views	186
19.3	Using String Views as Parameters	186
19.3.1	String View Considered Harmful	188
19.4	String View Types and Operations	192
19.4.1	Concrete String View Types	192
19.4.2	String View Operations	193
19.4.3	String View Support by Other Types	196
19.5	Using String Views in APIs	196
19.5.1	Using String Views instead of Strings	197
19.6	Afternotes	198
20	The Filesystem Library	201
20.1	Basic Examples	201
20.1.1	Print Attributes of a Passed Filesystem Path	201
20.1.2	Switch Over Filesystem Types	204
20.1.3	Create Different Types of Files	206
20.1.4	Dealing with Filesystems Using Parallel Algorithms	210
20.2	Principles and Terminology	210
20.2.1	General Portability Disclaimer	210
20.2.2	Namespace	211
20.2.3	Paths	211
20.2.4	Normalization	212
20.2.5	Member Function versus Free-Standing Functions	213
20.2.6	Error Handling	214

20.2.7	File Types	216
20.3	Path Operations	217
20.3.1	Path Creation	217
20.3.2	Path Inspection	218
20.3.3	Path I/O and Conversions	221
20.3.4	Conversions Between Native and Generic Format	224
20.3.5	Path Modifications	226
20.3.6	Path Comparisons	228
20.3.7	Other Path Operations	229
20.4	Filesystem Operations	230
20.4.1	File Attributes	230
20.4.2	File Status	234
20.4.3	Permissions	235
20.4.4	Filesystem Modifications	237
20.4.5	Symbolic Links and Filesystem-Dependent Path Conversions	241
20.4.6	Other Filesystem Operations	243
20.5	Iterating Over Directories	244
20.5.1	Directory Entries	246
20.6	Afternotes	248

Part IV: Library Extensions and Modifications 249

21	Extensions of Type Traits	251
21.1	Type Traits Suffix <code>_v</code>	251
21.2	New Type Traits	252
21.3	Afternotes	257
22	Parallel STL Algorithms	259
22.1	Using Parallel Algorithms	260
22.1.1	Using a Parallel <code>for_each()</code>	260
22.1.2	Using a Parallel <code>sort()</code>	263
22.2	Execution Policies	265
22.3	Exception Handling	266
22.4	Benefit of Not Using Parallel Algorithms	266
22.5	Overview of Parallel Algorithms	267

22.6	Motivation for New Algorithms for Parallel Processing	269
22.6.1	reduce()	269
22.7	Afternotes	278
23	New STL Algorithms in Detail	279
23.1	std::for_each_n()	279
23.2	New Numeric STL Algorithms	281
23.2.1	std::reduce()	281
23.2.2	std::transform_reduce()	283
23.2.3	std::inclusive_scan() and std::exclusive_scan()	287
23.2.4	std::transform_inclusive_scan() and std::transform_exclusive_scan() 289	
23.3	Afternotes	291
24	Substring and Subsequence Searchers	293
24.1	Using Substring Searchers	293
24.1.1	Using Searchers with search()	293
24.1.2	Using Searchers Directly	295
24.2	Using General Subsequence Searchers	296
24.3	Using Searcher Predicates	297
24.4	Afternotes	298
25	Other Utility Functions and Algorithms	299
25.1	size(), empty(), and data()	299
25.1.1	Generic size() Function	299
25.1.2	Generic empty() Function	301
25.1.3	Generic data() Function	301
25.2	as_const()	302
25.2.1	Capturing by Const Reference	302
25.3	clamp()	303
25.4	sample()	304
25.5	Afternotes	307
26	Container and String Extensions	309
26.1	Node Handles	309
26.1.1	Modifying a Key	309

26.1.2	Moving Nodes Between Containers	311
26.1.3	Merging Containers	312
26.2	Emplace Improvements	314
26.2.1	Return Type of Emplace Functions	314
26.2.2	try_emplace() and insert_or_assign() for Maps	314
26.2.3	try_emplace()	314
26.2.4	insert_or_assign()	315
26.3	Container Support for Incomplete Types	316
26.4	String Improvements	318
26.5	Afternotes	319
27	Multi-Threading and Concurrency	321
27.1	Supplementary Mutexes and Locks	321
27.1.1	std::scoped_lock	321
27.1.2	std::shared_mutex	322
27.2	is_always_lock_free for Atomics	323
27.3	Cache Line Sizes	324
27.4	Afternotes	326
28	Other Small Library Features and Modifications	327
28.1	std::uncaught_exceptions()	327
28.2	Shared Pointer Improvements	329
28.2.1	Special handling for Shared Pointers to Raw C Arrays	329
28.2.2	reinterpret_pointer_cast for Shared Pointers	330
28.2.3	weak_type for Shared Pointers	330
28.2.4	weak_from_this for Shared Pointers	330
28.3	Numeric Extensions	332
28.3.1	Greatest Common Divisor and Least Common Multiple	332
28.3.2	Three-Argument Overloads of std::hypot()	332
28.3.3	Mathematical Special Functions	332
28.4	chrono Extensions	334
28.5	constexpr Extensions and Fixes	335
28.6	noexcept Extensions and Fixes	336
28.7	Afternotes	336

Part V: Expert Utilities	339
29 Polymorphic Memory Resources (PMR)	341
29.1 Using Standard Memory Resources	342
29.1.1 Motivating Example	342
29.1.2 Standard Memory Resources	347
29.1.3 Standard Memory Resources in Detail	349
29.2 Defining Custom Memory Resources	355
29.2.1 Equality of Memory Resources	358
29.3 Providing Memory Resource Support for Custom Types	360
29.3.1 Definition of a PMR Type	360
29.3.2 Using a PMR Type	362
29.3.3 Dealing with the Different Types	363
29.4 Afternotes	364
30 new and delete with Over-Aligned Data	365
30.1 Using new with Alignments	365
30.1.1 Distinct Dynamic/Heap Memory Arenas	366
30.1.2 Passing the Alignment with the new Expression	367
30.2 Implementing operator new() for Aligned Memory	370
30.2.1 Implementing Aligned Allocation Before C++17	370
30.2.2 Implementing Type-Specific operator new()	372
30.3 Implementing Global operator new()	378
30.3.1 Backward Incompatibilities	379
30.4 Tracking All ::new Calls	380
30.5 Afternotes	383
31 std::to_chars() and std::from_chars()	385
31.1 Motivation for Low-Level Conversions between Character Sequences and Numeric Values	385
31.2 Example Usage	386
31.2.1 from_chars()	386
31.2.2 to_chars()	387
31.3 Floating-Point Round-Trip Support	388
31.4 Afternotes	391

32	std::launder()	393
32.1	Motivation for std::launder()	393
32.2	How launder() Solves the Problem	396
32.3	Why/When launder() Does Not Work	397
32.4	Afternotes	398
33	Improvements for Implementing Generic Code	399
33.1	std::invoke<>()	399
33.2	std::bool_constant<>	401
33.3	std::void_t<>	403
33.4	Afternotes	404
Part VI:	Final General Hints	405
34	Common C++17 Settings	407
34.1	Value of __cplusplus	407
34.2	Compatibility to C11	407
34.3	Dealing with Signal Handlers	408
34.4	Forward Progress Guarantees	408
34.5	Afternotes	408
35	Deprecated and Removed Features	409
35.1	Deprecated and Removed Core Language Features	409
35.1.1	Throw Specifications	409
35.1.2	Keyword register	409
35.1.3	Disable ++ for bool	410
35.1.4	Trigraphs	410
35.1.5	Definition/Redeclaration of static constexpr Members	410
35.2	Deprecated and Removed Library Features	410
35.2.1	auto_ptr	410
35.2.2	Algorithm random_shuffle()	410
35.2.3	unary_function and binary_function	411
35.2.4	ptr_fun(), mem_fun(), and Binders	411
35.2.5	Allocator Support for std::function<>	411
35.2.6	Deprecated IOStream Aliases	411

35.2.7	Deprecated Library Features	412
35.3	Afternotes	412
Glossary		415
Index		417