



C++

CORE GUIDELINES EXPLAINED



RAINER GRIMM

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



C++ Core Guidelines Explained

This page intentionally left blank



C++ Core Guidelines Explained

Best Practices for Modern C++

Rainer Grimm

◆ Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover image: SVPanteon/Shutterstock

Author photo on page xxix: © Karin Ruider

Cippi illustrations on pages 3, 7, 15, 27, 53, 131, 139, 165, 213, 231, 279, 293, 301, 375, 383, 397: © Beatrix Jaud-Grimm

Figure 5.2: © Howard Hinnant

Figure 9.7: © Matt Godbolt

Figures 4.2, 4.3, 8.11, 9.11, 10.13, 10.16, 12.1, 16.9, A.1-A.4: © Microsoft Corporation 2021

Figures 3.1, 4.3-4.8, 5.2-5.20, 6.1, 7.1-7.4, 7.6, 7.7, 8.1-8.9, 8.11-8.14, 9.1-9.5, 10.5-10.11, 10.13, 10.14, 10.16, 10.17, 13.1-13.11, 13.13-13.17, 13.24-13.27, 14.1-14.4, 15.1-15.4, 16.1-16.8, 18.1, 18.2, A.5, A.6: Screenshot of Konsole © KDE

Figures 10.2, 10.3: Screenshot of ThreadSanitizer © Google LLC

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com

Library of Congress Control Number: 2022930162

Copyright © 2022 Pearson Education, Inc.

The C++ Core Guidelines are copyright © Standard C++ Foundation and its contributors.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions/.

ISBN-13: 978-0-13-687567-3

ISBN-10: 0-13-687567-X

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at <https://www.pearson.com/report-bias.html>.

This page intentionally left blank

Contents

List of selected C++ Core Guidelines	xiii
List of figures	xxiii
List of tables	xxvii
Foreword	xxix
Preface	xxxii
Acknowledgments	xxxvii
About the author	xxxix
Part I: The Guidelines	1
Chapter 1: Introduction	3
Target readership	3
Aim	4
Non-aims	4
Enforcement	4
Structure	4
Major sections	5
Chapter 2: Philosophy	7
Chapter 3: Interfaces	15
The curse of non-const global variables	16
Dependency injection as a cure	18
Making good interfaces	20
Related rules	25
Chapter 4: Functions	27
Function definitions	28
Good names	28
Parameter passing: in and out	32
Parameter passing: ownership semantics	38

Value return semantics	42
When to return a pointer (τ^*) or an lvalue reference ($\tau\&$)	42
Other functions	46
Lambdas	46
Related rules	52
Chapter 5: Classes and Class Hierarchies	53
Summary rules	54
Concrete types	58
Constructors, assignments, and destructors	59
Default operations	60
Constructor	66
Copy and move	78
Destructors	83
Other default operations	88
Class hierarchies	98
General rules	99
Designing classes	102
Accessing objects	114
Overloading and overloaded operators	117
Conventional usage	118
Unions	126
Related rules	129
Chapter 6: Enumerations	131
General rules	131
Related rules	137
Chapter 7: Resource Management	139
General rules	140
Allocation and deallocation	145
Smart pointers	150
Basic usage	150
Function parameters	156
Related rules	164
Chapter 8: Expressions and Statements	165
General	166
Declarations	168
Names	168
Variables and their initialization	175

Macros	184
Expressions	186
Complicated expressions	186
Pointers	191
Order of evaluation	194
Conversions	197
Statements	199
Iteration statements	199
Selection statements	201
Arithmetic	204
Arithmetic with signed/unsigned integers	204
Typical arithmetic errors	208
Related rules	210
Chapter 9: Performance	213
Wrong optimizations	214
Wrong assumptions	214
Enable optimization	218
Related rules	230
Chapter 10: Concurrency	231
General guidelines	232
Concurrency	245
Locks	246
Threads	250
Condition variables	254
Data sharing	257
Resources	261
Overlooked danger	264
Parallelism	266
Message passing	269
Sending a value, or an exception	270
Sending a notification	272
Lock-free programming	273
Related rules	277
Chapter 11: Error Handling	279
Design	281
Communication	281
Invariants	282

Implementation	283
Do's	283
Don'ts	286
If you can't throw	288
Related rules	292
Chapter 12: Constants and Immutability	293
Use const	294
Use constexpr	298
Chapter 13: Templates and Generic Programming	301
Use	302
Interfaces	305
Advantages of function objects	307
Definition	320
Alternative implementations with specializations	325
Hierarchies	330
Variadic templates	332
Perfect forwarding	333
Variadic templates	335
Metaprogramming	336
Template metaprogramming	337
Type-traits library	345
Constant expressions	356
Other rules	362
Related rules	372
Chapter 14: C-Style Programming	375
Entire source code available	376
Entire source code not available	378
Chapter 15: Source Files	383
Interface and implementation files	384
Namespaces	391
Chapter 16: The Standard Library	397
Containers	398
Text	404
Input and output	411
Related rules	419

Part II: Supporting Sections	421
Chapter 17: Architectural Ideas	423
Chapter 18: Nonrules and Myths	427
Chapter 19: Profiles	437
Pro.typeType safety	438
Pro.boundsBounds safety	439
Pro.lifetimeLifetime safety	439
Chapter 20: Guidelines Support Library	441
Views	441
Ownership pointers	442
Assertions	443
Utilities	443
Part III: Appendixes	445
Appendix A: Enforcing the C++ Core Guidelines	447
Visual Studio	448
clang-tidy	450
Appendix B: Concepts	453
Appendix C: Contracts	457
Index	459

This page intentionally left blank

List of selected C++ Core Guidelines

P.1	Express ideas directly in code	8
P.2	Write in ISO Standard C++	8
P.3	Express intent	9
P.4	Ideally, a program should be statically type safe	10
P.5	Prefer compile-time checking to run-time checking	10
P.6	What cannot be checked at compile-time should be checkable at run-time	11
P.7	Catch run-time errors early	11
P.8	Don't leak any resources	11
P.9	Don't waste time or space	11
P.10	Prefer immutable data to mutable data	12
P.11	Encapsulate messy constructs, rather than spreading through the code	12
P.12	Use supporting tools as appropriate	13
P.13	Use support libraries as appropriate	13
I.2	Avoid non-const global variables	16
I.3	Avoid singletons	17
I.13	Do not pass an array as a single pointer	22
I.27	For stable library ABI, consider the Pimpl idiom	23
F.4	If a function may have to be evaluated at compile-time, declare it constexpr	29
F.6	If your function may not throw, declare it noexcept	30
F.8	Prefer pure functions	31
F.15	Prefer simple and conventional ways of passing information	32
F.16	For "in" parameters, pass cheaply-copied types by value and others by reference to const	34

E.19	For “forward” parameters, pass by <code>TP&&</code> and only <code>std::forward</code> the parameter	34
E.17	For “in-out” parameters, pass by reference to non-const	36
E.20	For “out” output values, prefer return values to output parameters	36
E.21	To return multiple “out” values, prefer returning a struct or tuple	37
E.42	Return a <code>T*</code> to indicate a position (only)	42
E.44	Return a <code>T&</code> when copy is undesirable and “returning no object” isn’t needed	42
E.45	Don’t return a <code>T&&</code>	44
E.48	Don’t return <code>std::move(local)</code>	44
E.46	<code>int</code> is the return type for <code>main()</code>	45
E.50	Use a lambda when a function won’t do (to capture local variables, or to write a local function)	46
E.52	Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms	47
E.53	Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread	48
E.51	Where there is a choice, prefer default arguments over overloading	49
E.55	Don’t use <code>va_arg</code> arguments	49
C.1	Organize related data into structures (structs or classes)	54
C.2	Use <code>class</code> if the class has an invariant; use <code>struct</code> if the data members can vary independently	55
C.3	Represent the distinction between an interface and an implementation using a class	56
C.4	Make a function a member only if it needs direct access to the representation of a class	56
C.5	Place helper functions in the same namespace as the class they support	57
C.7	Don’t define a <code>class</code> or <code>enum</code> and declare a variable of its type in the same statement	57
C.8	Use <code>class</code> rather than <code>struct</code> if any member is non-public	58
C.9	Minimize exposure of members	58
C.10	Prefer concrete types over class hierarchies	59
C.11	Make concrete types regular	59
C.20	If you can avoid defining any default operations, do	60
C.21	If you define or <code>=delete</code> any default operation, define or <code>=delete</code> them all	61
C.22	Make default operations consistent	63

C.41	A constructor should create a fully initialized object	67
C.42	If a constructor cannot construct a valid object, throw an exception	68
C.43	Ensure that a copyable (value type) class has a default constructor	69
C.45	Don't define a default constructor that only initializes data members; use member initializers instead	69
C.46	By default, declare single-argument constructors <code>explicit</code>	72
C.47	Define and initialize member variables in the order of member declaration	74
C.48	Prefer in-class initializers to member initializers in constructors for constant initializers	75
C.49	Prefer initialization to assignment in constructors	76
C.51	Use delegating constructors to represent common actions for all constructors of a class	76
C.52	Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization	77
C.67	A polymorphic class should suppress copying	81
C.30	Define a destructor if a class needs an explicit action at object destruction	84
C.31	All resources acquired by a class must be released by the class's destructor	84
C.32	If a class has a raw pointer (τ^*) or reference ($\tau\&$), consider whether it might be owning	85
C.33	If a class has an owning pointer member, define a destructor	85
C.35	A base class destructor should be either public and virtual, or protected and non-virtual	86
C.80	Use <code>=default</code> if you have to be explicit about using the default semantics	89
C.81	Use <code>=delete</code> when you want to disable default behavior (without wanting an alternative)	90
C.82	Don't call virtual functions in constructors and destructors	91
C.86	Make <code>==</code> symmetric with respect to operand types and <code>noexcept</code>	95
C.87	Beware of <code>==</code> on base classes	97
C.120	Use class hierarchies to represent concepts with inherent hierarchical structure (only)	99
C.121	If a base class is used as an interface, make it an abstract class	101

C.122	Use abstract classes as interfaces when complete separation of interface and implementation is needed	101
C.126	An abstract class typically doesn't need a constructor	102
C.128	Virtual functions should specify exactly one of <code>virtual</code> , <code>override</code> , or <code>final</code>	102
C.130	For making deep copies of polymorphic classes prefer a virtual <code>clone</code> function instead of copy construction/assignment	103
C.132	Don't make a function <code>virtual</code> without reason	105
C.131	Avoid trivial getters and setters	106
C.133	Avoid <code>protected</code> data	106
C.134	Ensure all non- <code>const</code> data members have the same access level	107
C.129	When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance	107
C.135	Use multiple inheritance to represent multiple distinct interfaces	111
C.138	Create an overload set for a derived class and its bases with <code>using</code>	111
C.140	Do not provide different default arguments for a virtual function and an overrider	113
C.146	Use <code>dynamic_cast</code> where class hierarchy navigation is unavoidable	115
C.147	Use <code>dynamic_cast</code> to a reference type when failure to find the required class is considered an error	115
C.148	Use <code>dynamic_cast</code> to a pointer type when failure to find the required class is considered a valid alternative	115
C.152	Never assign a pointer to an array of derived class objects to a pointer to its base	117
C.167	Use an operator for an operation with its conventional meaning	118
C.161	Use nonmember functions for symmetric operators	118
C.164	Avoid implicit conversion operators	122
C.162	Overload operations that are roughly equivalent	124
C.163	Overload only for operations that are roughly equivalent	124
C.168	Define overloaded operators in the namespace of their operands	125
C.180	Use unions to save memory	126
C.181	Avoid "naked" unions	127
C.182	Use anonymous unions to implement tagged unions	128
Enum.1	Prefer enumerations over macros	132
Enum.2	Use enumerations to represent sets of related named constants	133
Enum.3	Prefer enum classes over "plain" enums	133
Enum.5	Don't use <code>ALL_CAPS</code> for enumerators	134

Enum.6	Avoid unnamed enumerations	134
Enum.7	Specify the underlying type of an enumeration only when necessary	135
Enum.8	Specify enumerator values only when necessary	136
R.1	Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)	140
R.3	A raw pointer (a T*) is non-owning	143
R.4	A raw reference (a T&) is non-owning	143
R.5	Prefer scoped objects, don't heap-allocate unnecessarily	143
R.10	Avoid malloc() and free()	145
R.11	Avoid calling new and delete explicitly	146
R.12	Immediately give the result of an explicit resource allocation to a manager object	147
R.13	Perform at most one explicit resource allocation in a single expression statement	148
R.20	Use unique_ptr or shared_ptr to represent ownership	150
R.21	Prefer unique_ptr over shared_ptr unless you need to share ownership	151
R.22	Use make_shared() to make shared_ptrs	153
R.23	Use make_unique() to make unique_ptrs	153
R.24	Use std::weak_ptr to break cycles of shared_ptrs	154
R.30	Take smart pointers as parameters only to explicitly express lifetime semantics	157
R.37	Do not pass a pointer or reference obtained from an aliased smart pointer	162
ES.1	Prefer the standard library to other libraries and to "handcrafted code"	166
ES.2	Prefer suitable abstractions to direct use of language features	167
ES.5	Keep scopes small	168
ES.6	Declare names in for-statement initializers and conditions to limit scope	168
ES.7	Keep common and local names short, and keep uncommon and nonlocal names longer	169
ES.8	Avoid similar-looking names	170
ES.9	Avoid ALL_CAPS names	170
ES.10	Declare one name (only) per declaration	171
ES.11	Use auto to avoid redundant repetition of type names	171
ES.12	Do not reuse names in nested scopes	172
ES.20	Always initialize an object	175

ES.21	Don't introduce a variable (or constant) before you need to use it . . .	176
ES.22	Don't declare a variable until you have a value to initialize it with . . .	176
ES.23	Prefer the {}-initializer syntax	177
ES.26	Don't use a variable for two unrelated purposes	182
ES.28	Use lambdas for complex initialization, especially of const variables	183
ES.40	Avoid complicated expressions	186
ES.41	If in doubt about operator precedence, parenthesize	187
ES.42	Keep use of pointers simple and straightforward	187
ES.45	Avoid "magic constants"; use symbolic constants	190
ES.55	Avoid the need for range checking	190
ES.47	Use <code>nullptr</code> rather than <code>0</code> or <code>NULL</code>	191
ES.61	Delete arrays using <code>delete[]</code> and non-arrays using <code>delete</code>	194
ES.65	Don't dereference an invalid pointer	194
ES.43	Avoid expressions with undefined order of evaluation	194
ES.44	Don't depend on order of evaluation of function arguments	195
ES.48	Avoid casts	197
ES.49	If you must use a cast, use a named cast	198
ES.50	Don't cast away <code>const</code>	199
ES.78	Don't rely on implicit fallthrough in <code>switch</code> statements	201
ES.79	Use <code>default</code> to handle common cases (only)	202
ES.100	Don't mix signed and unsigned arithmetic	204
ES.101	Use unsigned types for bit manipulation	205
ES.102	Use signed types for arithmetic	205
ES.106	Don't try to avoid negative values by using <code>unsigned</code>	206
ES.103	Don't overflow	208
ES.104	Don't underflow	208
ES.105	Don't divide by zero	210
Per.7	Design to enable optimization	219
Per.10	Rely on the static type system	222
Per.11	Move computation from run time to compile-time	223
Per.19	Access memory predictably	225
CP.1	Assume that your code will run as part of a multi-threaded program	232
CP.2	Avoid data races	234
CP.3	Minimize explicit sharing of writable data	236
CP.4	Think in terms of tasks, rather than threads	237

CP.8	Don't try to use <code>volatile</code> for synchronization	238
CP.9	Whenever feasible use tools to validate your concurrent code	238
CP.20	Use RAII, never plain <code>lock()/unlock()</code>	246
CP.21	Use <code>std::lock()</code> or <code>std::scoped_lock</code> to acquire multiple mutexes	247
CP.22	Never call unknown code while holding a lock (e.g., a callback)	249
CP.23	Think of a joining thread as a scoped container	250
CP.24	Think of a thread as a global container	250
CP.25	Prefer <code>std::jthread</code> over <code>std::thread</code>	251
CP.26	Don't <code>detach()</code> a thread	253
CP.42	Don't wait without a condition	254
CP.31	Pass small amounts of data between threads by value, rather than by reference or pointer	257
CP.32	To share ownership between unrelated threads use <code>shared_ptr</code>	258
CP.40	Minimize context switching	261
CP.41	Minimize thread creation and destruction	261
CP.43	Minimize time spent in a critical section	264
CP.44	Remember to name your <code>lock_guards</code> and <code>unique_locks</code>	264
CP.100	Don't use lock-free programming unless you absolutely have to	273
CP.101	Distrust your hardware/compiler combination	274
CP.102	Carefully study the literature	276
E.3	Use exceptions for error handling only	283
E.14	Use purpose-designed user-defined types as exceptions (not built-in types)	283
E.15	Catch exceptions from a hierarchy by reference	285
E.13	Never throw while being the direct owner of an object	286
E.30	Don't use exception specifications	287
E.31	Properly order your catch-clauses	288
Con.1	By default, make objects immutable	294
Con.2	By default, make member functions <code>const</code>	294
Con.3	By default, pass pointers and references to <code>const</code> s	297
Con.4	Use <code>const</code> to define objects with values that do not change after construction	297
Con.5	Use <code>constexpr</code> for values that can be computed at compile-time	298
T.1	Use templates to raise the level of abstraction of code	302
T.2	Use templates to express algorithms that apply to many argument types	304
T.3	Use templates to express containers and ranges	305

T.40	Use function objects to pass operations to algorithms	305
T.42	Use template aliases to simplify notation and hide implementation details	310
T.43	Prefer using over typedef for defining aliases	311
T.44	Use function templates to deduce class template argument types (where feasible)	311
T.46	Require template arguments to be at least Regular or SemiRegular	313
T.47	Avoid highly visible unconstrained templates with common names	314
T.48	If your compiler does not support concepts, fake them with enable_if	319
T.60	Minimize a template's context dependencies	320
T.61	Do not over-parameterize members	321
T.62	Place non-dependent class template members in a non-templated base class	323
T.80	Do not naively template a class hierarchy	331
T.83	Do not declare a member function template virtual	331
T.140	Name all operations with potential for reuse	362
T.141	Use an unnamed lambda if you need a simple function object in one place only	364
T.143	Don't write unintentionally nongeneric code	365
T.144	Don't specialize function templates	367
CPL.1	Prefer C++ to C	375
CPL.2	If you must use C, use the common subset of C and C++, and compile the C code as C++	376
CPL.3	If you must use C for interfaces, use C++ in the calling code using such interfaces	378
SF.1	Use a .cpp suffix for code files and .h for interface files if your project doesn't already follow another convention	384
SF.2	A .h file may not contain object definitions or non-inline function definitions	385
SF.5	A .cpp file must include the .h file(s) that defines its interface	386
SF.8	Use #include guards for all .h files	388
SF.9	Avoid cyclic dependencies among source files	388
SF.10	Avoid dependencies on implicitly #included names	390
SF.11	Header files should be self-contained	391

SF.6	Use <code>using namespace</code> directives for transition, for foundation libraries (such as <code>std</code>), or within a local scope (only)	391
SF.7	Don't write <code>using namespace</code> at global scope in a header file	393
SF.20	Use namespaces to express logical structure	394
SF.21	Don't use an unnamed (anonymous) namespace in a header	394
SF.22	Use an unnamed (anonymous) namespace for all internal/nonexported entities	394
SL.con.1	Prefer using STL array or vector instead of a C-array	398
SL.con.2	Prefer using STL vector by default unless you have a reason to use a different container	402
SL.con.3	Avoid bounds errors	403
SL.str.1	Use <code>std::string</code> to own character sequences	405
SL.str.2	Use <code>std::string_view</code> to refer to character sequences	407
SL.str.4	Use <code>char*</code> to refer to a single character	409
SL.str.5	Use <code>std::byte</code> to refer to byte values that do not necessarily represent characters	409
SL.str.12	Use the <code>s</code> suffix for string literals meant to be standard-library strings	410
SL.io.1	Use character-level input only when you have to	411
SL.io.2	When reading, always consider ill-formed input	411
SL.io.3	Prefer <code>iostreams</code> for I/O	413
SL.io.10	Unless you use <code>printf</code> -family functions call <code>ios_base::sync_with_stdio(false)</code>	414
SL.io.50	Avoid <code>endl</code>	415
A.1	Separate stable code from less stable code	423
A.2	Express potentially reusable parts as a library	424
A.4	There should be no cycles among libraries	425
NR.1	Don't insist that all declarations should be at the top of a function	427
NR.2	Don't insist to have only a single return-statement in a function	428
NR.3	Don't avoid exceptions	429
NR.4	Don't insist on placing each class declaration in its own source file	431
NR.5	Don't use two-phase initialization	431
NR.6	Don't place all cleanup actions at the end of a function and <code>goto exit</code>	433
NR.7	Don't make all data members protected	436

This page intentionally left blank

List of figures

3.1	Dependency injection	20
4.1	Assembler instructions to the program <code>constexpr.cpp</code>	30
4.2	Returning a <code>std::pair</code>	38
4.3	The five ownership semantics	41
4.4	Displaying arbitrary characters	44
4.5	Causing a core dump	44
4.6	Returning a reference to a temporary	45
4.7	Summation with <code>va_arg</code>	51
4.8	Summation with fold expressions	52
5.1	Automatically generated special member functions	61
5.2	Double free detected with <code>AddressSanitizer</code>	63
5.3	Output of <code>strange.cpp</code>	66
5.4	Directly initializing in the class	71
5.5	Converting constructor	73
5.6	Wrong initialization order of member variables	75
5.7	Slicing	83
5.8	A class with a <code>std::unique_ptr</code>	86
5.9	<code>delete</code> the destructor	91
5.10	Calling a virtual function in the constructor	92
5.11	A virtual <code>clone</code> member function	105
5.12	A virtual <code>clone</code> member function without covariant return type	105
5.13	Shadowing of member functions	112
5.14	Different default arguments for virtual functions	114
5.15	<code>dynamic_cast</code> causes a <code>std::bad_cast</code> exception	116
5.16	Missing overload for <code>int</code> and <code>MyInt</code>	119
5.17	Using an <code>explicit</code> constructor	121
5.18	Implicit operator <code>bool</code>	124
5.19	Explicit operator <code>bool</code>	124
5.20	Undefined behavior with a “naked” union	128
6.1	The enumerators are too big for the underlying type	136

7.1	Resource Acquisition Is Initialization	142
7.2	Undefined behavior causes a core dump	146
7.3	Two owners with <code>std::unique_ptr</code>	148
7.4	Moving a <code>std::unique_ptr</code>	153
7.5	Cycles of smart pointers	154
7.6	Cycles of smart pointers	156
7.7	Lifetime semantics of smart pointers	159
8.1	Reusing names in nested scopes	173
8.2	Hiding member functions of a base	174
8.3	Change visibility with a <code>using</code> declaration	175
8.4	The most vexing parse	180
8.5	Narrowing conversion	181
8.6	Narrowing conversion detected	182
8.7	Usage of the function-like macro <code>max</code>	185
8.8	The null pointers <code>0</code> , <code>NULL</code> , and <code>nullptr</code>	193
8.9	Unspecified behavior	196
8.10	Wrong casts with the Visual Studio compiler	197
8.11	Wrong casts with the GCC or Clang compiler	198
8.12	Modulo versus overflow with <code>unsigneds</code> and <code>signeds</code>	208
8.13	Detecting narrowing conversion	208
8.14	Underflow and overflow of a C-array	209
9.1	Performance of the Meyers singleton	217
9.2	Performance of the singleton based on acquire-release semantics	218
9.3	Performance of the singleton in the single-threaded case	218
9.4	Move semantics on a copy-only type	221
9.5	Invoking <code>gcd</code> at compile time and run time	225
9.6	The relevant assembler instructions to the algorithm <code>gcd</code>	225
9.7	<code>std::deque</code>	226
9.8	<code>std::list</code>	226
9.9	<code>std::forward_list</code>	226
9.10	Memory access for sequence containers on Windows	229
10.1	Four categories of variables	235
10.2	Data race detection with ThreadSanitizer	241
10.3	Overview of <code>CppMem</code>	242
10.4	A data race in <code>CppMem</code>	245
10.5	A deadlock due to multiple locked mutexes	248
10.6	Forgot to join a thread	252

10.7	A condition variable in action	255
10.8	A condition variable without a predicate	257
10.9	Shared ownership using a pointer	259
10.10	Shared ownership using a smart pointer	261
10.11	Thread creation on Linux	263
10.12	Thread creation on Windows	263
10.13	Using a temporary lock <code>std::lock_guard</code>	266
10.14	Using a named temporary <code>std::lock_guard</code>	266
10.15	Usage of <code>std::transform_exclusive_scan</code>	269
10.16	A value and an exception as message	271
10.17	Notifications with a task	273
12.1	A <code>mutable</code> variable	296
13.1	A function, a function object, and a lambda as sorting criteria	307
13.2	A function object with state	309
13.3	A lambda with state	310
13.4	Template argument deduction	313
13.5	A reference is not <code>SemiRegular</code>	315
13.6	Surprise with argument-dependent lookup	317
13.7	Surprises with argument-dependent lookup solved	319
13.8	<code>std::enable_if</code>	320
13.9	Comparing two accounts	326
13.10	Comparing two accounts with a binary predicate	330
13.11	Compiler error with a virtual member function	332
13.12	Calculating primes at compile time	338
13.13	Calculating the factorial of 5 at compile time	340
13.14	Calculating at run time and compile time	343
13.15	<code>power</code> as function and metafunction	345
13.16	Type comparisons	350
13.17	Correctness with the type-traits functions	354
13.18	Function versus template arguments	359
13.19	Modification versus new value	359
13.20	Recursion versus loop	360
13.21	Template specialization for conditional execution	360
13.22	Update versus new value	361
13.23	Simulating a return value	361
13.24	Case-insensitive search in a <code>struct</code>	363
13.25	Iterating through a few containers	366

13.26	Specialization and overloading of function templates	370
13.27	Specialization of function templates	372
14.1	Warnings with a C compiler	376
14.2	Errors with a C++ compiler	377
14.3	Different size of a char with a C++ compiler	377
14.4	Function overloading	379
15.1	Multiple definitions of a function	386
15.2	Linker error because of mismatch between function declaration and definition	387
15.3	Compiler error because of a mismatch between function declaration and definition	387
15.4	Cyclic dependencies among source files	389
16.1	Automatic management of memory	400
16.2	sizeof a C-array, a C++-array, and a std::vector	402
16.3	Accessing a nonexistent element of a std::string	404
16.4	Undefined behavior with a C-string	406
16.5	No memory allocation with std::string_view	408
16.6	Undefined behavior with printf	414
16.7	Performance with/without flushing on Linux	418
16.8	Performance with/without flush on Windows	418
18.1	Different return types in a function	429
18.2	Automatic managing of a device	435
A.1	Enable code analysis	448
A.2	Configure the applied rules	449
A.3	Automatic managing of a device	449
A.4	Suppress warnings	450
A.5	Check the C++ Core Guidelines exclusively	451
A.6	Check the C++ Core Guidelines with clang-tidy	452

List of tables

4.1	Normal parameter passing	32
4.2	Advanced parameter passing	33
4.3	Ownership semantics of parameter passing	38
7.1	Smart pointers as function parameters	156
10.1	Typical thread size	262
10.2	Algorithms of the STL for which parallel versions are available (the std namespace is omitted)	267
10.3	Condition variables versus tasks	272
10.4	Operation reordering on various platforms	276
13.1	Comparing two accounts	330
13.2	Composite type categories	348
13.3	Template metaprogramming versus constexpr functions	358
13.4	Iterator categories	366
14.1	Name mangling	380
16.1	Various kinds of text	405
16.2	State of the stream	412

This page intentionally left blank

Foreword

C++ is a very rich, very expressive language with lots of features. It has to be because a successful general-purpose programming language must have more facilities than any one developer needs, and a living and evolving language will accumulate alternative idioms for expressing an idea. That can lead to choice overload. So, what does a developer choose for programming style and mastery? How does a developer avoid getting stuck with outdated and ineffective techniques and programming styles?

The C++ Core Guidelines (<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>) are an ongoing open-source project to address such issues by gathering widely recognized modern C++ best practices together in one place. The Core Guidelines rely on decades of experience and earlier sets of coding rules. They share a conceptual framework with C++ itself, with a focus on type safety, resource safety, and the elimination of avoidable complexities and inefficiencies. The Core Guidelines are organized to address known problem areas and partly phrased to enable enforcement by a static analyzer.

The Core Guidelines are organized as a reference work to make it easy to look up and share specific topics, not as a tutorial to be read sequentially to learn themes for using modern C++ well. We are therefore very pleased to see Rainer Grimm applying his teaching skills and industrial background to tackle the hard and necessary task of making the rules accessible to more people. We hope that you find learning the Core Guidelines stimulating and, especially, that applying them to your real-world problems will make your work significantly more effective and more enjoyable.

Bjarne Stroustrup

Herb Sutter

This page intentionally left blank

Preface

This preface serves one purpose: to give you, dear reader, the necessary background to get the most out of this book. This background includes technical details about me, my writing style, my motivation for writing this book, and the challenges of writing such a book. If you want to skip this section, fine, but at least read the Acknowledgments section.

Conventions

I promise, only a few conventions.

Rules versus guidelines

The authors of the C++ Core Guidelines often refer to them as rules. So do I. In the context of this book, I use both terms interchangeably.

Special fonts

- | | |
|------------------|--|
| Bold | Sometimes I use bold font to emphasize important terms. |
| <i>Italic</i> | Italics designate hyperlinks (eBook only). |
| Monospace | Code, instructions, keywords, names of types, variables, functions, and classes are displayed in monospace font. |

Boxes

I use boxes with a bullet list for the information concluding each chapter.

Related rules

Often rules are related to other rules. I provide this valuable information at the end of the chapter if necessary.

Distilled

Important

Get the essential information at the end of each chapter.

Source code

I dislike using directives and declarations because they hide the origin of the library functions. Due to the limited length of a page, I have to use them from time to time. I use them in such a way that the origin can always be deduced from the using directive (`using namespace std;`) or the using declaration (`using std::cout;`). Not all headers are displayed in the code snippets. Boolean values are displayed with `true` or `false`. The necessary I/O manipulator `std::boolalpha` is mostly not part of the code snippets.

Three dots (`...`) in the code snippets stand for missing code.

When I present a complete program as a code example, you will find the name of the source file in the first line of the code. I assume that you use a C++14 compiler. If the example needs C++17 or C++20 support, I mention the required C++ standard after the name.

I often use markers such as `// (1)` in the source file to ease my explanations. If possible, I write the marker in the cited line or, if not, one line before. The markers are not part of the more than 100 source files that are part of the book (available from <https://github.com/RainerGrimm/CppCoreGuidelines>). For layout reasons, I often adjusted the source code in this book.

When I use examples from the C++ Core Guidelines, I often rewrite them for readability by adding `namespace std` if it is missing, or unify the layout.

Why guidelines?

This subjective observation is mainly based on my more than 15 years of experience as a trainer for C++, Python, and software development in general. In the last few years, I was responsible for the team and the software deployed on defibrillators. My responsibility included regulatory affairs for our devices. Writing software for a defibrillator is extremely challenging because they can cause death or serious injury for the patient and the operator.

I have a question in mind that we should answer as a C++ community. This question is: Why do we need guidelines for modern C++? Here are my thoughts, which consist of three observations.

Complex for novices

C++ is, in particular for beginners, an inherently complex language. This is mainly because the problems we want to solve are inherently complicated and often complex

as well. When you teach C++, you should provide a set of rules that work for your participants in at least 95% of all use cases. I think about rules such as

- Let the compiler deduce your types.
- Initialize with curly braces.
- Prefer tasks over threads.
- Use smart pointers instead of raw pointers.

I teach rules such as the ones mentioned in my seminars. We need a canon of best practices or rules in C++. These rules should be formulated positively and not negatively. They should declare how you should write code and not what should be avoided.

Challenging for professionals

I'm not worried about the sheer amount of new features that we get with each new C++ standard every three years. I'm worried about the new ideas that modern C++ supports. Think about event-driven programming with coroutines, lazy evaluation, infinite data streams, or function composition with the ranges library. Think about concepts, which introduce semantic categories to template parameters. It can be quite challenging to teach C programmers object-oriented ideas. When you shift, therefore, to these new paradigms, you have to rethink and presumably change the way you solve your programming challenges. I assume that this plethora of new ideas will, in particular, overwhelm professional programmers. They are the ones who are used to solving the problems with their classical techniques. With high probability, they fall into the *hammer-nail trap*.

Used in safety-critical software

In the end, I have a strong concern. In safety-critical software development, you often have to stick to guidelines. The most prominent are MISRA C++. The current MISRA C++:2008 guidelines were published by the *Motor Industry Software Reliability Association*. They are based on the *MISRA C guidelines* from the year 1998. Initially designed for the automotive industry, they became the de facto standard for the implementation of safety-critical software in the aviation, military, and medical sectors. As MISRA C, MISRA C++ describes guidelines for a safe subset of C++. But there is a conceptual problem. MISRA C++ is not state of the art for modern

software development in C++. It's four standards behind! Here is an example: MISRA C++ doesn't allow operator overloading. I teach in my seminars that you should use user-defined literals to implement type-safe arithmetic: `auto constexpr dist = 4 * 5_m + 10_cm - 3_dm`. To implement such type-safe arithmetic, you have to overload the arithmetic operators and the literal operators for the suffixes. To be honest, I don't believe that MISRA C++ will ever evolve in lockstep with the current C++ standard. Only community-driven guidelines such as the C++ Core Guidelines can face this challenge.

MISRA C++ integrates AUTOSAR C++14

However, there is hope. MISRA C++ integrates AUTOSAR C++14. AUTOSAR C++14 is based on C++14 and should become an extension of the MISRA C++ standard. I'm highly skeptical that organization-driven guidelines can keep in lockstep with the dynamics of modern C++.

My challenge

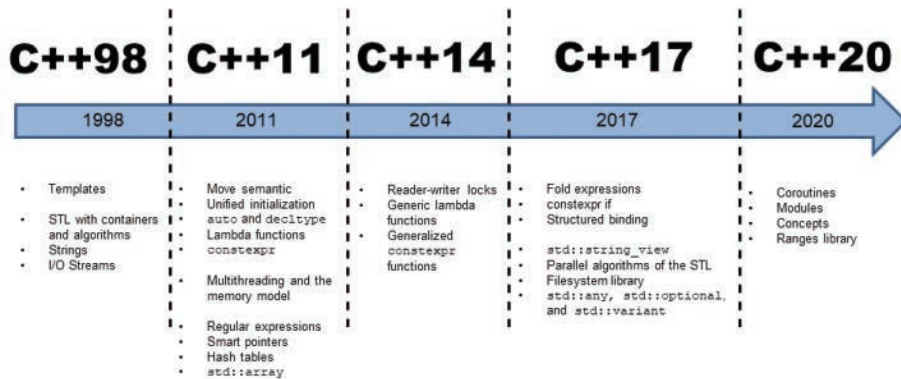
Let me share the essential lines of my e-mail discussion in May 2019 with Bjarne Stroustrup and Herb Sutter telling them that I wanted to write a book about the C++ Core Guidelines: "I'm an absolute fan of the value which is inside the C++ Core Guidelines because my strong belief is that we need guidelines for the correct/safe usage of modern C++. I often use examples or ideas from the C++ Core Guidelines in my C++ classes. The format reminds me of the MISRA C++ or AUTOSAR C++14 rules which is presumably intentional, but this is not the ideal format for a big audience. I think that more people would read and reason about the guidelines if we had a second document which describes the general ideas of the guidelines."

I want to add a few remarks to these previous conversations. In the last few years, I wrote on my German and English blogs more than a hundred posts about the C++ Core Guidelines. Additionally, I write for the German *Linux-Magazin* a series on the C++ Core Guidelines. I do this for two reasons: First, the C++ Core Guidelines should become better known, and second, I want to present them in a readable form, extended with background information if necessary.

Here is my challenge: The C++ Core Guidelines consist of over five hundred guidelines, most of the time just called rules. These rules are designed with static analysis in mind. Many of the rules are lifesaving for a professional C++ software developer, but also many of the rules are quite special, often incomplete or redundant, and sometimes the rules even contradict. My challenge is to boil these valuable rules down to a readable, even entertaining, story, removing the esoteric stuff and filling the gaps if necessary. In the end, the book should contain the rules that are mandatory for a professional software developer in C++.

Panta rhei

Panta rhei, or “everything flows,” from the Greek philosopher *Heraclitus* stands for the challenge I’m faced with while writing this book. The C++ Core Guidelines are a *GitHub-hosted* project with more than 200 contributors. While I was writing this book, the source I was basing my writing on may have changed.



The guidelines already include C++ features, which may become part of an upcoming standard, such as contracts in C++23. To reflect this challenge, I made a few decisions.

1. I provide links in the electronic version of this book to the mentioned C++ Core Guidelines so you can quite easily refer to their origins.
2. My focus is on the C++17 standard. If appropriate, I include guidelines targeting the C++20 standard, such as concepts.
3. The C++ Core Guidelines evolve constantly, in particular as new C++ standards are published. So will this book. My plan is to update this book accordingly.

How to read this book

The structure of this book represents the structure of the C++ Core Guidelines. It has the corresponding major sections and parts of the supporting sections. In addition to the C++ Core Guidelines, I included appendixes, which provide concise overviews of missing topics, including C++20 or even C++23 features.

I still have not answered one question: how to read this book. Of course, you should start with the major sections, best from top to toe. The supporting sections provide additional information and introduce, in particular, the Guidelines Support Library. Use the appendixes as a kind of reference to get the necessary background information to understand the major sections. Without this additional information, this book would not be complete.

Acknowledgments

First of all, I have to thank all contributors to the C++ Core Guidelines. The Core Guidelines are the work of about 250 contributors; the most prolific so far have been Herb Sutter, Bjarne Stroustrup, Gabriel Dos Reis, Sergey Zubkov, Jonathan Wakely, and Neil MacIntosh (Guidelines Support Library). If you want to know all other contributors, go to <https://github.com/isocpp/CppCoreGuidelines/graphs/contributors>.

Second, I want to thank my proofreaders very much. Without their help, the book would not have the quality it has now. Here are their names in alphabetic order: Yaser Afshar, Nicola Bombace, Sylvain Dupont, Fabio Fracassi, Juliette Grimm, Michael Möllney, Mateusz Nowak, Arthur O'Dwyer, and Moritz Strübe.

Third, many thanks to my wife, Beatrix Jaud-Grimm, for drawing the illustrations for this book.

This page intentionally left blank

About the author



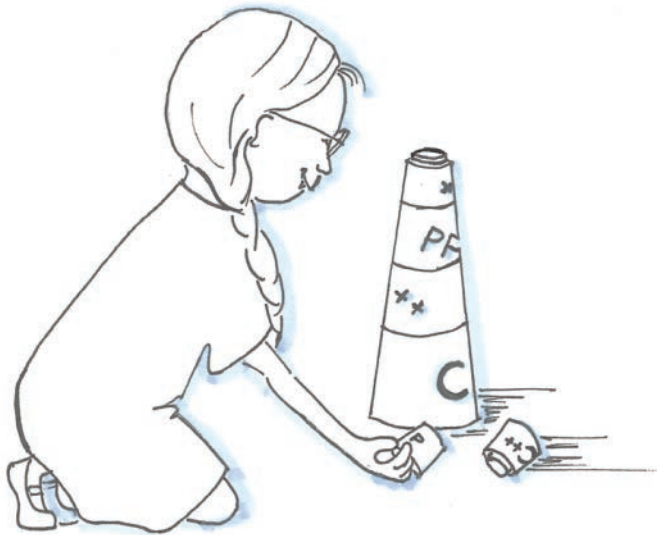
I have worked as a software architect, team lead, and instructor since 1999. In 2002, I created a continuing education program at my company. I have given seminars since 2002. My first seminars were about proprietary management software, but seminars for Python and C++ followed immediately. In my spare time, I like to write articles about C++, Python, and Haskell. I also like to speak at conferences. I publish weekly in English and German on my blog *Modernes Cpp*, hosted by *Heise Developer*.

Since 2016, I have been an independent instructor giving seminars about modern C++ and Python. I have published several books in various languages about modern C++ and concurrency, in particular. Due to my profession, I always search for the best way to teach modern C++.

This page intentionally left blank

Chapter 3

Interfaces



Cippi assembles components.

An interface is a contract between a service provider and a service user. *Interfaces* are, according to the C++ Core Guidelines, “probably the most important single aspect of code organization.” The section on interfaces has about twenty rules. Four of the rules are related to contracts, which didn’t make it into the C++20 standard.

A few rules related to interfaces involve contracts, which may be part of C++23. A contract specifies preconditions, postconditions, and invariants for functions that

can be checked at run time. Due to the uncertainty of the future, I ignore these rules. The appendix provides a short introduction to contracts.

Let me end this introduction with my favorite quote from Scott Meyers:

Make interfaces easy to use correctly and hard to use incorrectly.

I.2

Avoid non-const global variables

Of course, you should avoid non-const global variables. But why? Why is a global variable, in particular when it is non-constant, bad? A global injects a hidden dependency into the function, which is not part of the interface. The following code snippet makes my point:

```
int glob{2011};

int multiply(int fac) {
    glob *= glob;
    return glob * fac;
}
```

The execution of the function `multiply` changes, as a side effect, the value of the global variable `glob`. Therefore, you cannot test the function or reason about the function in isolation. When more threads use `multiply` concurrently, you have to protect the variable `glob`. There are more drawbacks to non-const global variables. If the function `multiply` had no side effects, you could have stored the previous result and reused the cached value for performance reasons.

The curse of non-const global variables

Using non-const globals has many drawbacks. First and foremost, non-const globals break encapsulation. This breaking of encapsulation makes it impossible to think about your functions/classes (entities) in isolation. The following bullet points enumerate the main drawbacks of non-const global variables.

- **Testability:** You cannot test your entities in isolation. There are no units, and therefore, there is no unit testing. You can only perform system testing. The effect of your entities depends on the state of the entire system.

- **Refactoring:** It is quite challenging to refactor your code because you cannot reason about your code in isolation.
- **Optimization:** You cannot easily rearrange the function invocations or perform the function invocations on different threads because there may be hidden dependencies. It's also extremely dangerous to cache previous results of function calls.
- **Concurrency:** The necessary condition for having a data race is a shared, mutable state. Non-const global variables are shared and mutable.

I.3

Avoid singletons

Sometimes, global variables are very well disguised.

```
// singleton.cpp

class MySingleton {

public:
    MySingleton(const MySingleton&)= delete;
    MySingleton& operator = (const MySingleton&)= delete;

    static MySingleton* getInstance() {
        if ( !instance ){
            instance= new MySingleton();
        }
        return instance;
    }

private:
    static MySingleton* instance;
    MySingleton()= default;
    ~MySingleton()= default;
};

MySingleton* MySingleton::instance= nullptr;

int main() {

    std::cout << MySingleton::getInstance() << "\n";
```

```
std::cout << MySingleton::getInstance() << "\n";  
  
}
```

A singleton is just a global, and you should, therefore, *avoid singletons*, if possible. A singleton gives the straightforward guarantee that only one instance of a class exists. As a global, a singleton injects a dependency, which ignores the interface of a function. This is due to the fact that singletons as static variables are typically invoked directly: `Singleton::getInstance()` as shown in the two lines of the main function. The direct invocation of the singleton has a few serious consequences. You cannot *unit test* a function having a singleton because there is no unit. Additionally, you cannot fake your singleton and replace it during run time because the singleton is not part of the function interface. To make it short: Singletons break the testability of your code.

Implementing a singleton seems like a piece of cake but is not. You are faced with a few challenges:

- Who is responsible for destroying the singleton?
- Should it be possible to derive from the singleton?
- How can you initialize a singleton in a thread-safe way?
- In which sequence are singletons initialized when they depend on each other and are in different translation units? This is to scare you. This challenge is called the *static initialization order problem*.

The bad reputation of the singleton is, in particular, due to an additional fact. Singletons were heavily overused. I see programs that consist entirely of singletons. There are no objects because the developer wants to prove that they apply design patterns.

Dependency injection as a cure

When an object uses a singleton, it injects a hidden dependency into the object. Thanks to dependency injection, this dependency is part of the interface, and the service is injected from the outside. Consequently, there is no dependency between the client and the injected service. Typical ways to inject dependencies are constructors, setter members, or template parameters.

The following program shows how you can replace a logger using dependency injection.

```
// dependencyInjection.cpp

#include <chrono>
#include <iostream>
#include <memory>

class Logger {
public:
    virtual void write(const std::string&) = 0;
    virtual ~Logger() = default;
};

class SimpleLogger: public Logger {
    void write(const std::string& mess) override {
        std::cout << mess << std::endl;
    }
};

class TimeLogger: public Logger {
    using MySecondTick = std::chrono::duration<long double>;
    long double timeSinceEpoch() {
        auto timeNow = std::chrono::system_clock::now();
        auto duration = timeNow.time_since_epoch();
        MySecondTick sec(duration);
        return sec.count();
    }
    void write(const std::string& mess) override {
        std::cout << std::fixed;
        std::cout << "Time since epoch: " << timeSinceEpoch()
    }
};

class Client {
public:
    Client(std::shared_ptr<Logger> log): logger(log) {}
    void doSomething() {
        logger->write("Message");
    }
    void setLogger(std::shared_ptr<Logger> log) {
        logger = log;
    }
};
```

```
private:
    std::shared_ptr<Logger> logger;
};

int main() {

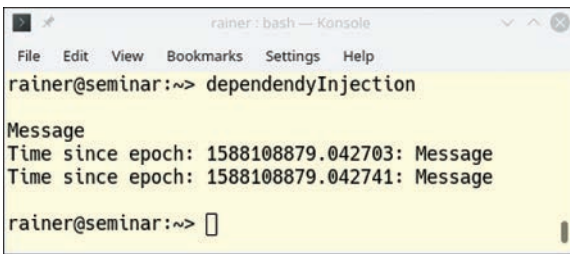
    std::cout << '\n';

    Client c1(std::make_shared<SimpleLogger>()); // (1)
    c1.doSomething();
    c1.setLogger(std::make_shared<TimeLogger>()); // (2)
    c1.doSomething();
    c1.doSomething();

    std::cout << '\n';

}
```

The client `c1` supports the constructor (1) and the member function `setLogger` (2) to inject the logger service. In contrast to the `SimpleLogger`, the `TimeLogger` includes the time since epoch in its message (see Figure 3.1).



```
rainer: bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> dependencyInjection

Message
Time since epoch: 1588108879.042703: Message
Time since epoch: 1588108879.042741: Message

rainer@seminar:~> █
```

Figure 3.1 *Dependency injection*

Making good interfaces

Functions should not communicate via global variables but through interfaces. Now we are in the core of this chapter. According to the C++ Core Guidelines, here are the recommendations for interfaces. Interfaces should follow these rules:

- Make interfaces explicit (I.1).
- Make interfaces precise and strongly typed (I.4).

- Keep the number of function arguments low (I.23).
- Avoid adjacent unrelated parameters of the same type (I.24).

The first function `showRectangle` breaks all mentioned rules for interfaces:

```
void showRectangle(double a, double b, double c, double d) {
    a = floor(a);
    b = ceil(b);

    ...
}
```

```
void showRectangle(Point top_left, Point bottom_right);
```

Although the first function `showRectangle` should show only a rectangle, it modifies its arguments. Essentially, it has two purposes and has, as a consequence, a misleading name (I.1). Additionally, the function signature does not provide any information about what the arguments should be, nor in which sequence the arguments must be given (I.23 and I.24). Furthermore, the arguments are doubles without a constraint value range. This constraint must, therefore, be established in the function body (I.4). In contrast, the second function `showRectangle` takes two concrete points. Checking to see if a `Point` has valid value is the job of the constructor of `Point`. This responsibility should not be the job of the function.

I want to elaborate more on the rules *I.23* and *I.24* and the function `std::transform_reduce` from the Standard Template Library (STL). First, I need to define the term callable. A callable is something that behaves like a function. This can be a function but also a function object, or a lambda expression. If a callable accepts one argument, it is called a unary callable; if it takes two arguments, it is called a binary callable.

`std::transform_reduce` first applies a unary callable to one range or a binary callable to two ranges and then a binary callable to the resulting range. When you use `std::transform_reduce` with a unary lambda expression, the call is easy to use correctly:

```
std::vector<std::string> strVec{"Only", "for", "testing", "purpose"};

std::size_t res = std::transform_reduce(
    std::execution::par,
    strVec.begin(), strVec.end(),
    0,
    [](std::size_t a, std::size_t b) { return a + b; },
```



```

    [](std::string s) { return s.size(); }
);

```

The function `std::transform_reduce` transforms each string onto its length (`[](const std::string s) { return s.size(); }`) and applies the binary callable (`[](std::size_t a, std::size_t b) { return a + b; }`) to the resulting range. The initial value for the summation is 0. The whole calculation is performed in parallel: `std::execution::par`.

When you use the overload, which accepts two binary callables, the declaration of the function becomes quite complicated and error prone. Consequently, it breaks the rules *I.23* and *I.24*.

```

template<class ExecutionPolicy,
        class ForwardIt1, class ForwardIt2, class T,
        class BinaryOp1, class BinaryOp2>
T transform_reduce(ExecutionPolicy&& policy,
                 ForwardIt1 first1, ForwardIt1 last1,
                 ForwardIt2 first2,
                 T init, BinaryOp1 binary_op1, BinaryOp2 binary_op2);

```

Calling this overload would require six template arguments and seven function arguments. Using the binary callables in the correct sequence may also be a challenge.

```
transform | reduce
```

The main reason for the complicated function `std::transform_reduce` is that two functions are combined into one. Defining two separate functions `transform` and `reduce` and supporting function composition via the pipe operator would be a better choice: `transform | reduce`.

I.13

Do not pass an array as a single pointer

The guideline that you should not pass an array as a single pointer is special. I can tell you from experience that this rule is a common cause of undefined behavior. For instance, the function `copy_n` is quite error prone.

```

template <typename T>
void copy_n(const T* p, T* q, int n); // copy from [p:p+n) to [q:q+n)

...

```

```
int a[100] = {0, };
int b[100] = {0, };

copy_n(a, b, 101);
```

Maybe you had an exhausting day and you miscounted by one. The result is an off-by-one error and, therefore, undefined behavior. The cure is simple. Use a container from the STL such as `std::vector` and check the size of the container in the function body. C++20 offers `std::span`, which solves this issue more elegantly. A `std::span` is an object that can refer to a contiguous sequence of objects. A `std::span` is never an owner. This contiguous memory can be an array, a pointer with a size, or a `std::vector`.

```
template <typename T>
void copy(std::span<const T> src, std::span<T> des);

int arr1[] = {1, 2, 3};
int arr2[] = {3, 4, 5};

...

copy(arr1, arr2);
```

`copy` doesn't need the number of elements. Hence, a common cause of errors is eliminated with `std::span<T>`.

1.27

For stable library ABI, consider the Pimpl idiom

An application binary interface (ABI) is the interface between two binary programs.

Thanks to the PImpl idiom, you can isolate the users of a class from its implementation and, therefore, avoid recompilation. PImpl stands for pointer to implementation and is a programming technique in C++ that removes implementation details from a class by placing them in a separate class. This separate class is accessed by a pointer. This is done because private data members participate in class layout and private member functions participate in overload resolution. These dependencies mean that changes to those implementation details require recompilation of all users of a class. A class holding a pointer to implementation (PImpl) can isolate the users of a class from changes in its implementation at the cost of an indirection.

The C++ Core Guidelines show a typical implementation.

- **Interface:** `Widget.h`

```
class Widget {
    class impl;
    std::unique_ptr<impl> pimpl;
public:
    void draw(); // public API that will be forwarded
                // to the implementation
    Widget(int); // defined in the implementation file
    ~Widget();  // defined in the implementation file,
                // where impl is a complete type
    Widget(Widget&&) = default;
    Widget(const Widget&) = delete;
    Widget& operator = (Widget&&); // defined in the
                                  // implementation file
    Widget& operator = (const Widget&) = delete;
};
```

- **Implementation:** `Widget.cpp`

```
class Widget::impl {
    int n; // private data
public:
    void draw(const Widget& w) { /* ... */ }
    impl(int n) : n(n) {}
};
void Widget::draw() { pimpl->draw(*this); }
Widget::Widget(int n) : pimpl{std::make_unique<impl>(n)} {}
Widget::~Widget() = default;
Widget& Widget::operator = (Widget&&) = default;
```

cppreference.com provides more information about the PImpl idiom. Additionally, the rule “C.129: When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance” shows how to apply the PImpl idiom to dual inheritance.

Related rules

I present the rule “I.10: Use exceptions to signal a failure to perform a required task” in Chapter 11, Error Handling, the rule “I.11: Never transfer ownership by a raw pointer (τ^*) or reference ($\tau\&$)” in Chapter 4, Functions, the rule “I.22: Avoid complex initialization of global objects” in Chapter 8, Expressions and Statements, and the rule “I.25: Prefer abstract classes as interfaces to class hierarchies” in Chapter 5, Classes and Class Hierarchies.

Distilled

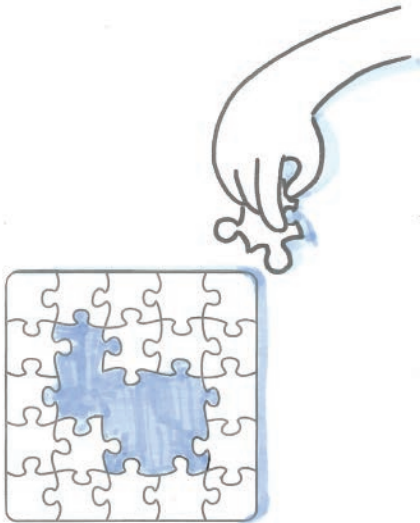
Important

- Don't use global variables. They introduce hidden dependencies.
- Singletons are global variables in disguise.
- Interfaces and in particular functions should express their intent.
- Interfaces should be strongly typed and have few arguments that cannot be easily confused.
- Don't take a C-array by pointer but use a `std::span`.
- If you want to separate the users of a class from its implementation, use the PImpl idiom.

This page intentionally left blank

Chapter 4

Functions



Cippi uses functions to solve the challenge.

Software developers master complexity by dividing complex tasks into smaller units. After the small units are addressed, they put the smaller units together to master the complex task. A function is a typical unit and, therefore, the basic building block for a program. Functions are “the most critical part in most interfaces . . .” (C++ Core Guidelines about functions).

The C++ Core Guidelines have about forty rules for functions. They provide valuable information on the definition of functions, how you should pass the arguments (e.g., by copy or by reference), and what that means for the ownership semantics. They also state rules about the semantics of the return value and other functions such as lambdas. Let's dive into them.

Function definitions

Presumably, the most important principle for good software is good names. This principle is often ignored and holds true in particular for functions.

Good names

The C++ Core Guidelines dedicate the first three rules to good names: “F.1: ‘Package’ meaningful operations as carefully named functions,” “F.2: A function should perform a single logical operation,” and “F.3: Keep functions short and simple.”

Let me start with a short anecdote. A few years ago, a software developer asked me, “How should I call my function?” I told him to give the function a name such as `verbObject`. In case of a member function, a `verb` may be fine because the function already operates on an object. The verb stands for the operation that is performed on the object. The software developer replied that this is not possible; the function must be called `getTimeAndAddToPhonebook` or just `processData` because the functions perform more than one job (single-responsibility principle). When you don't find a meaningful name for your function (F.1), that's a strong indication that your function does more than one logical operation (F.2) and that your function isn't short and simple (F.3). A function is too long if it does not fit on a screen. A screen means roughly 60 lines by 140 characters, but your measure may differ. Now you should identify the operations of the function and package these operations into carefully named functions.

The guidelines present an example of a bad function:

```
void read_and_print() { // bad
    int x;
    std::cin >> x;
    // check for errors
    std::cout << x << '\n';
}
```

The function `read_and_print` is bad for many reasons. The function is tied to a specific input and output and cannot be used in a different context. Refactoring the function into two functions solves these issues and makes it easier to test and to maintain:

```
int read(std::istream& is) { // better
    int x;
    is >> x;
    // check for errors
    return x;
}

void print(std::ostream& os, int x) {
    os << x << '\n';
}
```

F.4

If a function may have to be evaluated at compile-time, declare it `constexpr`

A `constexpr` function is a function that has the potential to run at compile time. When you invoke a `constexpr` function within a constant expression, or you take the result of a `constexpr` with a `constexpr` variable, it runs at compile time. You can invoke a `constexpr` function with arguments that can be evaluated only at run time, too. `constexpr` functions are implicit `inline`.

The result of `constexpr` evaluated at compile time is stored in the ROM (read-only memory). Performance is, therefore, the first big benefit of a `constexpr` function. The second is that `constexpr` functions evaluated at compile time are `const` and, therefore, thread safe.

Finally, a result of the calculation is made available at run time as a constant in ROM.

```
// constexpr.cpp

constexpr auto gcd(int a, int b) {
    while (b != 0) {
        auto t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```



```

}

int main() {

    constexpr int i = gcd(11, 121);    // (1)

    int a = 11;
    int b = 121;
    int j = gcd(a, b);                // (2)

}

```

Figure 4.1 shows the output of Compiler Explorer and depicts the assembly code generated by the compiler for this function. I used the Microsoft Visual Studio Compiler 19.22 without optimization.

```

32 main PROC
33 $LN3:
34 sub rsp, 56 ; 00000038H
35 mov DWORD PTR i$, 11
36 mov DWORD PTR a$, 11
37 mov DWORD PTR b$, 121 ; 00000079H
38 mov edx, DWORD PTR b$, [rsp]
39 mov ecx, DWORD PTR a$, [rsp]
40 call int gcd(int,int) ; gcd
41 mov DWORD PTR j$, [rsp], eax
42 xor eax, eax
43 add rsp, 56 ; 00000038H
44 ret 0
45 main ENDP

```

Figure 4.1 Assembler instructions to the program `constexpr.cpp`

Based on the colors, you can see that (1) in the source code corresponds to line 35 in the assembler instructions and (2) in the source code corresponds to lines 38–41 in the assembler instructions. The call `constexpr int i = gcd(11, 121);` boils down to the value 11, but the call `int j = gcd(a, b);` results in a function call.

F.6

If your function may not throw, declare it `noexcept`

By declaring a function as `noexcept`, you reduce the number of alternative control paths; therefore, `noexcept` is a valuable hint to the optimizer. Even if your function can

throw, noexcept often makes much sense. noexcept means in this case: I don't care. The reason may be that you have no way to react to an exception. Therefore, the only way to deal with exceptions is to invoke `std::terminate()`. This noexcept declaration is also a piece of valuable information for the reader of your code.

The next function just crashes if it runs out of memory.

```
std::vector<std::string> collect(std::istream& is) noexcept {
    std::vector<std::string> res;
    for (std::string s; is >> s;) {
        res.push_back(s);
    }
    return res;
}
```

The following types of functions should never throw: destructors (see the section Failing Destructor in Chapter 5), swap functions, move operations, and default constructors.

F.8

Prefer pure functions

Pure functions are functions that always return the same result when given the same arguments. This property is also called referential transparency. Pure functions behave like infinite big lookup tables.

The function template `square` is a pure function:

```
template<class T>
auto square(T t) {
    return t * t;
}
```

Conversely, impure functions are functions such as `random()` or `time()`, which can return a different result from call to call. To put it another way, functions that interact with state outside the function body are impure.

Pure functions have a few very interesting properties. You should, therefore, prefer pure functions, if possible.

Pure functions can

- Be tested in isolation
- Be verified or refactorized in isolation

- Cache their result
- Automatically be reordered or be executed on other threads

Pure functions are also often called mathematical functions. Functions in C++ are by default not pure such as in the pure functional programming language Haskell. Using pure functions is based on the discipline of the programmer. `constexpr` functions are pure when evaluated at compile time. Template metaprogramming is a pure functional language embedded in the imperative language C++.

Chapter 13, *Templates and Generic Programming*, gives a concise introduction to programming at compile time, including template metaprogramming.

Parameter passing: in and out

The C++ Core Guidelines have a few rules to express various ways to pass parameters in and out of functions.

F.15

Prefer simple and conventional ways of passing information

The first rule presents the big picture. First, it provides an overview of the various ways to pass information in and out of a function (see Table 4.1).

Table 4.1 *Normal parameter passing*

	Cheap to copy or impossible to copy	Cheap to move or moderate cost to move or don't know	Expensive to move
In	func(x)	func(const x&)	
In & retain "copy"			
In/Out	func(x&)		
Out	x func()		func(x&)

The table is very concise: The headings describe the characteristics of the data regarding the cost of copying and moving. The rows indicate the direction of parameter passing.

- Kind of data
 - **Cheap to copy or impossible to copy:** `int` or `std::unique_ptr`

- **Cheap to move:** `std::vector<T>` or `std::string`
- **Moderate cost to move:** `std::array<std::vector>` or `BigPOD` (POD stands for Plain Old Data—that is, a class without constructors, destructors, and virtual member functions.)
- **Don't know:** `template`
- **Expensive to move:** `BigPOD[]` or `std::array<BigPOD>`
- Direction of parameter passing
 - **In:** input parameter
 - **In & retain “copy”:** caller retains its copy
 - **In/Out:** parameter that is modified
 - **Out:** output parameter

A cheap operation is an operation with a few ints; moderate cost is about one thousand bytes without memory allocation.

These normal parameter passing rules should be your first choice. However, there are also advanced parameter passing rules (see Table 4.2). Essentially, the case with the “in & move from” semantics was added.

Table 4.2 *Advanced parameter passing*

	Cheap to copy or impossible to copy	Cheap to move or moderate cost to move or don't know	Expensive to move
In	func(X)	func(constX&)	
In & retain “copy”			
In & move from	func(X&&)		
In/Out	func(X&)		
Out	X func()		func(X&)

After the “in & move from” call, the argument is in the so-called moved-from state. Moved-from means that it is in a valid but not nearer specified state. Essentially, you have to initialize the moved-from object before using it again.

The remaining rules to parameter passing provide the necessary background information for these tables.

F.16

For “in” parameters, pass cheaply-copied types by value and others by reference to const

The rule is straightforward to follow. Input values should be copied by default if possible. When they cannot be cheaply copied, take them by const reference. The C++ Core Guidelines give a rule of thumb to the question, Which objects are cheap to copy or expensive to copy?

- You should pass a parameter `par` by value if `sizeof(par) < 3 * sizeof(void*)`.
- You should pass a parameter `par` by const reference if `sizeof(par) > 3 * sizeof(void*)`.

```
void f1(const std::string& s); // OK: pass by reference to const;
                             // always cheap

void f2(std::string s);      // bad: potentially expensive

void f3(int x);              // OK: unbeatable

void f4(const int& x);       // bad: overhead on access in f4()
```

F.19

For “forward” parameters, pass by `TP&&` and only `std::forward` the parameter

This rule stands for a special input value. Sometimes you want to forward the parameter `par`. This means an lvalue is copied and an rvalue is moved. Therefore, the constness of an lvalue is ignored and the rvalueness of an rvalue is preserved.

The typical use case for forwarding parameters is a factory function that creates an arbitrary object by invoking its constructor. You do not know if the arguments are rvalues nor do you know how many arguments the constructor needs.

```
// forwarding.cpp

#include <string>
```

```
#include <utility>

template <typename T, typename ... T1> // (1)
T create(T1&& ... t1) {
    return T(std::forward<T1>(t1)...);
}

struct MyType {
    MyType(int, double, bool) {}
};

int main() {

    // lvalue
    int five=5;
    int myFive= create<int>(five);

    // rvalues
    int myFive2= create<int>(5);

    // no arguments
    int myZero= create<int>();

    // three arguments; (lvalue, rvalue, rvalue)
    MyType myType = create<MyType>(myZero, 5.5, true);

}
```

The three dots (ellipsis) in the function `create` (1) denote a parameter pack. We call a template using a parameter pack a variadic template.

Packing and unpacking of the parameter pack

When the ellipsis is on the left of the type parameter `T1`, the parameter pack is packed; when on the right, it is unpacked. This unpacking in the return statement `T(std::forward<T1>(t1)...) essentially means that the expression std::forward<T1>(t1) is repeated until all arguments of the parameter pack are consumed and a comma is put between each subexpression. For the curious, C++ Insights shows this unpacking process.`

The combination of forwarding together with variadic templates is the typical creation pattern in C++. Here is a possible implementation of `std::make_unique<T>`.

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
std::make_unique<T> creates a std::unique_ptr for T
```

F.17

For “in-out” parameters, pass by reference to non-const

The rule communicates its intention to the caller: This function modifies its argument.

```
std::vector<int> myVec{1, 2, 3, 4, 5};

void modifyVector(std::vector<int>& vec) {
    vec.push_back(6);
    vec.insert(vec.end(), {7, 8, 9, 10});
}
```

F.20

For “out” output values, prefer return values to output parameters

The rule is straightforward. Just return the value, but don’t use a const value because it has no added value and interferes with move semantics. Maybe you think that copying a value is an expensive operation. Yes and no. Yes, you are right, but no, the compiler applies RVO (Return Value Optimization) or NRVO (Named Return Value Optimization). RVO means that the compiler is allowed to remove unnecessary copy operations. What was a possible optimization step becomes in C++17 a guarantee.

```
MyType func() {
    return MyType{};           // no copy with C++17
}
MyType myType = func();      // no copy with C++17
```

Two unnecessary copy operations can happen in these few lines, the first in the return call and the second in the function call. With C++17, no copy operation takes place. If the return value has a name, we call it NRVO. Maybe you guessed that.

```
MyType func() {
    MyType myValue;
    return myValue;           // one copy allowed
}
MyType myType = func();      // no copy with C++17
```

The subtle difference is that the compiler can still copy the value `myValue` in the return statement according to C++17. But no copy will take place in the function call.

Often, a function has to return more than one value. Here, the rule F.21 kicks in.

F.21

To return multiple “out” values, prefer returning a struct or tuple

When you insert a value into a `std::set`, overloads of the member function `insert` return a `std::pair` of an iterator to the inserted element and a `bool` set to true if the insertion was successful. `std::tie` with C++11 or structured binding with C++17 are two elegant ways to bind both values to a variable.

```
// returnPair.cpp; C++17

#include <iostream>
#include <set>
#include <tuple>

int main() {

    std::cout << '\n';

    std::set<int> mySet;

    std::set<int>::iterator iter;
    bool inserted = false;
    std::tie(iter, inserted) = mySet.insert(2011); // (1)
    if (inserted) std::cout << "2011 was inserted successfully\n";

    auto [iter2, inserted2] = mySet.insert(2017); // (2)
```



```

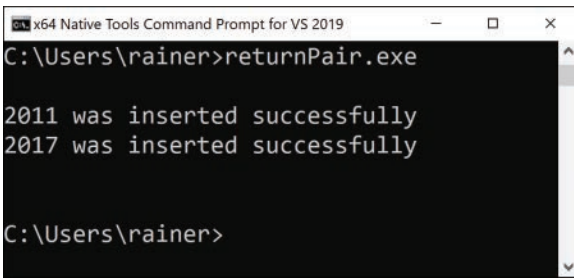
    if (inserted2) std::cout << "2017 was inserted successfully\n";

    std::cout << '\n';

}

```

Line (1) uses `std::tie` to unpack the return value of `insert` into `iter` and `inserted`. Line (2) uses structured binding to unpack the return value of `insert` into `iter2` and `inserted2`. `std::tie` needs, in contrast to structured binding, a predeclared variable. See Figure 4.2.



```

C:\Users\rainer>returnPair.exe

2011 was inserted successfully
2017 was inserted successfully

C:\Users\rainer>

```

Figure 4.2 *Returning a `std::pair`*

Parameter passing: ownership semantics

The last section was about the flow of parameters: which parameters are input, input/output, or output values. But there is more to arguments than the direction of the flow. Passing parameters is about ownership semantics. This section presents five typical ways to pass parameters: by copy, by pointer, by reference, by `std::unique_ptr`, or by `std::shared_ptr`. Only the rules to smart pointers are inside this section. The rule to pass by copy is part of the previous section Parameter Passing: In and Out, and the rules to pointers and references are part of Chapter 3, Interfaces.

Table 4.3 provides the first overview.

Table 4.3 *Ownership semantics of parameter passing*

Example	Ownership	Rule
<code>func(value)</code>	<code>func</code> is a single owner of the resource.	E.16
<code>func(pointer*)</code>	<code>func</code> has borrowed the resource.	I.11 and F.7
<code>func(reference&)</code>	<code>func</code> has borrowed the resource.	I.11 and F.7
<code>func(std::unique_ptr)</code>	<code>func</code> is a single owner of the resource.	E.26
<code>func(std::shared_ptr)</code>	<code>func</code> is a shared owner of the resource.	E.27

Here are more details:

- **func(value)**: The function `func` has its own copy of the `value` and is its owner. `func` automatically releases the resource.
- **func(pointer*)**: `func` has borrowed the resource and is, therefore, not authorized to delete the resource. `func` has to check before each usage that the pointer is not a null pointer.
- **func(reference&)**: `func` has borrowed the resource. In contrast to the pointer, the reference always has a valid value.
- **func(std::unique_ptr)**: `func` is the new owner of the resource. The caller of the `func` has explicitly transferred the ownership of the resource to the callee. `func` automatically releases the resource.
- **func(std::shared_ptr)**: `func` is an additional owner of the resource. `func` extends the lifetime of the resource. At the end of `func`, `func` ends its ownership of the resource. This end causes the release of the resource if `func` was the last owner.

Who is the owner?

It's very important to indicate ownership clearly. Just imagine that your program is written in legacy C++, and you have only a raw pointer at your disposal to express the four kinds of ownership by pointer, by reference, by `std::unique_ptr`, or by `std::shared_ptr`. The key question in legacy C++ is, Who is the owner?

The following code snippet makes my point:

```
void func(double* ptr) {  
    ...  
}  
  
double* ptr = new double[];  
func(ptr);
```

The critical question is, Who is the owner of the resource? The callee of `func` that uses the array, or the caller of the `func` that created the array? If `func` is the owner, it has to release the resource. If not, `func` is not allowed to release the resource. This condition is not satisfactory. If `func` does not release the resource, a memory leak may happen. If `func` does release the resource, undefined behavior may be the result.

In consequence, ownership needs to be documented. Defining the contract using the type system in modern C++ is a big step in the right direction to eliminate this ambiguity in documentation.

Using `std::move` on application level is not about moving. Using `std::move` on application level is about the transfer of ownership—for example, applying `std::move` to a `std::unique_ptr` transfers the ownership of the memory to another `std::unique_ptr`. The smart pointer `uniquePtr1` is the original owner, but `uniquePtr2` becomes the new owner.

```
auto uniquePtr1 = std::make_unique<int>(2011);
std::unique_ptr<int> uniquePtr2{ std::move(uniquePtr1) };
```

Here are five variants of ownership semantics in practice.

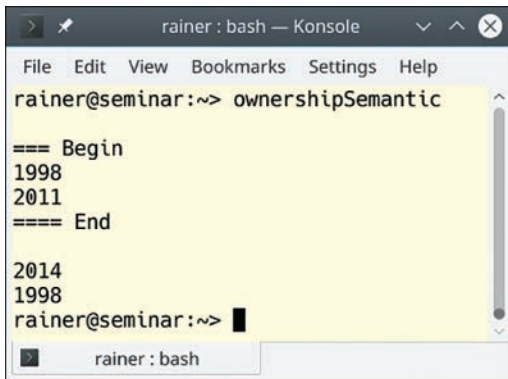
```
1 // ownershipSemantic.cpp
2
3 #include <iostream>
4 #include <memory>
5 #include <utility>
6
7 class MyInt {
8 public:
9     explicit MyInt(int val): myInt(val) {}
10    ~MyInt() noexcept {
11        std::cout << myInt << '\n';
12    }
13 private:
14    int myInt;
15 };
16
17 void funcCopy(MyInt myInt) {}
18 void funcPtr(MyInt* myInt) {}
19 void funcRef(MyInt& myInt) {}
20 void funcUniqPtr(std::unique_ptr<MyInt> myInt) {}
21 void funcSharedPtr(std::shared_ptr<MyInt> myInt) {}
22
23 int main() {
24
25    std::cout << '\n';
26
27    std::cout << "=== Begin" << '\n';
28
29    MyInt myInt{1998};
```

```

30     MyInt* myIntPtr = &myInt;
31     MyInt& myIntRef = myInt;
32     auto uniqPtr = std::make_unique<MyInt>(2011);
33     auto sharedPtr = std::make_shared<MyInt>(2014);
34
35     funcCopy(myInt);
36     funcPtr(myIntPtr);
37     funcRef(myIntRef);
38     funcUniqPtr(std::move(uniqPtr));
39     funcSharedPtr(sharedPtr);
40
41     std::cout << "==== End" << '\n';
42
43     std::cout << '\n';
44
45 }

```

The type `MyInt` displays in its destructor (lines 10–12) the value of `myInt` (line 14). The five functions in the lines 17–21 implement each of the ownership semantics. The lines 29–33 have the corresponding values. See Figure 4.3.



```

rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> ownershipSemantic

=== Begin
1998
2011
==== End

2014
1998
rainer@seminar:~>

```

Figure 4.3 *The five ownership semantics*

The screenshot shows that two destructors are called before and two destructors are called at the end of the `main` function. The destructors of the copied `myInt` (line 35) and the moved `uniquePtr` (line 38) are called before the end of `main`. In both cases, `funcCopy` or `funcUniqPtr` becomes the owner of the resource. The lifetime of the functions ends before the lifetime of `main`. This end of the lifetime does not hold for the original `myInt` (line 29) and the `sharedPtr` (line 33). Their lifetime ends with `main`, and therefore, the destructor is called at the end of the `main` function.

Value return semantics

The seven rules in this section are in accordance with the previously mentioned rule “F.20: For ‘out’ output values, prefer return values to output parameters.” The rules of this section are, in particular, about special use cases and don’ts.

When to return a pointer (T^*) or an lvalue reference ($T\&$)

As we know from the last section (Parameter Passing: Ownership Semantics), a pointer or a reference should never transfer ownership.

F.42

Return a T^* to indicate a position (only)

A pointer should indicate only a position. This is exactly what the function `find` does.

```
Node* find(Node* t, const string& s) {
    if (!t || t->name == s) return t;
    if ((auto p = find(t->left, s))) return p;
    if ((auto p = find(t->right, s))) return p;
    return nullptr;
}
```

The pointer indicates that the `Node` is holding the position of `s`.

F.44

Return a $T\&$ when copy is undesirable and “returning no object” isn’t needed

When return no object is not an option, using a reference instead of a pointer comes into play.

Sometimes you want to chain operations without unnecessary copying and destruction of temporaries. Typical use cases are input and output streams or assignment operators (“F.47: Return $T\&$ from assignment operators”). What is the subtle difference between returning by $T\&$ or returning by T in the following code snippet?

```
A& operator = (const A& rhs) { ... };
A operator = (const A& rhs) { ... };
```

```
A = a1, a2, a3;  
a1 = a2 = a3;
```

The copy assignment operator returning a copy (A) triggers the creation of two additional temporary objects of type A.

A reference to a local

Returning a reference (pointer) to a local is undefined behavior.

Undefined behavior essentially means this: Don't make any assumptions about your program. Fix undefined behavior. The program `lambdaFunctionCapture.cpp` returns a reference to a local.

```
// lambdaFunctionCapture.cpp  
  
#include <functional>  
#include <iostream>  
#include <string>  
  
auto makeLambda() {  
    const std::string val = "on stack created";  
    return [&val]{return val;};           // (2)  
}  
  
int main() {  
  
    auto bad = makeLambda();             // (1)  
    std::cout << bad();                 // (3)  
  
}
```

The `main` function calls the function `makeLambda()` (1). The function returns a lambda expression, which has a reference to the local variable `val` (2).

The call `bad()` (3) causes the undefined behavior because the lambda expression uses a reference to the local `val`. As local, its lifetime ends with the scope of `makeLambda()`.

Executing the program gives unpredictable results. Sometimes I get the entire string, sometimes a part of the string, or sometimes just the value 0. As an example, here are two runs of the program.

In the first run, arbitrary characters are displayed until the string terminating symbol (`\0`) ends it (see Figure 4.4).

T&&

You should not use a T&& as a return type. Here is a small example to demonstrate the issue.

```
// returnRvalueReference.cpp

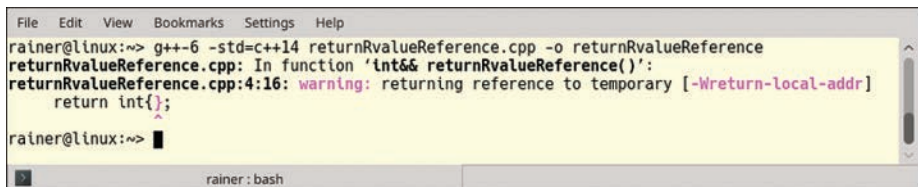
int&& returnRvalueReference() {
    return int{};
}

int main() {

    auto myInt = returnRvalueReference();

}
```

When compiled, the GCC compiler complains immediately about a reference to a temporary (see Figure 4.6). To be precise, the lifetime of the temporary ends with the end of the full expression `auto myInt = returnRvalueReference();`.



```
File Edit View Bookmarks Settings Help
rainer@linux:~> g++-6 -std=c++14 returnRvalueReference.cpp -o returnRvalueReference
returnRvalueReference.cpp: In function 'int&& returnRvalueReference()':
returnRvalueReference.cpp:4:16: warning: returning reference to temporary [-Wreturn-local-addr]
    return int{};
           ^
rainer@linux:~> █

rainer: bash
```

Figure 4.6 *Returning a reference to a temporary*

std::move(local)

Thanks to copy elision with RVO and NRVO, using `return std::move(local)` is not an optimization but a pessimization. Pessimization means that your program may become slower.

F.46

`int` is the return type for `main()`

According to the C++ standard, there are two variations of the `main` function:

```
int main() { ... }
int main(int argc, char** argv) { ... }
```


The second version is equivalent to `int main(int argc, char* argv[]) { ... }`.

The `main` function does not need a return statement. If control reaches the end of the `main` function without encountering a return statement, the effect is that of executing `return 0;`. `return 0` stands for the successful execution of the program.

Other functions

The rules in this section advise on when to use lambdas and compare `va_arg` with fold expressions.

Lambdas

F.50

Use a lambda when a function won't do (to capture local variables, or to write a local function)

This rule states the use case for lambdas. This immediately raises the question, When do you have to use a lambda or a function? Here are two obvious reasons.

1. If your callable has to capture local variables or is declared in a local scope, you have to use a lambda function.
2. If your callable should support overloading, use a function.

Now I want to present my crucial arguments for lambdas that are often ignored.

Expressiveness

“Explicit is better than implicit.” This meta-rule from Python (PEP 20—The Zen of Python) also applies to C++. It means that your code should explicitly express its intent (see rule “P.1: Express ideas directly in code”). Of course, this holds true in particular for lambdas.

```
std::vector<std::string> myStrVec = {"523345", "4336893456", "7234",
                                   "564", "199", "433", "2435345"};

std::sort(myStrVec.begin(), myStrVec.end(),
          [](const std::string& f, const std::string& s) {
              return f.size() < s.size();
            });
```

Compare this lambda with the function `lessLength`, which is subsequently used.

```
std::vector<std::string> myStrVec = {"523345", "4336893456", "7234",
                                   "564", "199", "433", "2435345"};

bool lessLength(const std::string& f, const std::string& s) {
    return f.size() < s.size();
}

std::sort(myStrVec.begin(), myStrVec.end(), lessLength);
```

Both the lambda and the function provide the same order predicate for the sort algorithm. Imagine that your coworker named the function `foo`. This means you have no idea what the function is supposed to do. As a consequence, you have to document the function.

```
// sorts the vector ascending, based on the length of its strings
std::sort(myStrVec.begin(), myStrVec.end(), foo);
```

Further, you have to hope that your coworker did it right. If you don't trust them, you have to analyze the implementation. Maybe that's not possible because you have the declaration of the function. With a lambda, your coworker cannot fool you. The code is the truth. Let me put it more provocatively: *Your code should be so expressive that it does not require documentation.*

Expressiveness versus don't repeat yourself (DRY)

The design rule to write expressive code with lambdas often contradicts another important design rule: Don't repeat yourself (DRY). DRY means that you should not write the same code more than once. Making a reusable unit such as a function and giving it a self-explanatory name is the appropriate cure for DRY. In the end, you have to decide in the concrete case if you rate expressiveness higher than DRY.

F.52

Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms

and

F.53

Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread

Both rules are strongly related, and they boil down to the following observation: A lambda should operate only on valid data. When the lambda captures the data by copy, the data is by definition valid. When the lambda captures data by reference, the lifetime of the data must outlive the lifetime of the lambda. The previous example with a reference to a local showed different results of a lambda referring to invalid data.

Sometimes the issue is not so easy to catch.

```
int main() {  
  
    std::string str{"C++11"};  
  
    std::thread thr([&str]{ std::cout << str << '\n'; });  
    thr.detach();  
  
}
```

Okay, I hear you say, “That is easy.” The lambda expression used in the created thread `thr` captures the variable `str` by reference. Afterward, `thr` is detached from the lifetime of its creator, which is the main thread. Therefore, there is no guarantee that the created thread `thr` uses a valid string `str` because the lifetime of `str` is bound to the lifetime of the main thread. Here is a straightforward way to fix the issue. Capture `str` by copy:

```
int main() {  
  
    std::string str{"C++11"};  
  
    std::thread thr([str]{ std::cout << str << '\n'; });  
    thr.detach();  
  
}
```

Problem solved? No! The crucial question is, Who is the owner of `std::cout`? `std::cout`'s lifetime is bound to the lifetime of the process. This means that the thread `thr` may be gone before `std::cout` prints `C++11` onscreen. The way to fix this

problem is to join the thread `thr`. In this case, the creator waits until the created is done, and therefore, capturing by reference is also fine.

```
int main() {  
  
    std::string str{"C++11"};  
  
    std::thread thr([&str]{ std::cout << str << '\n'; });  
    thr.join();  
  
}
```

F.51

Where there is a choice, prefer default arguments over overloading

If you need to invoke a function with a different number of arguments, prefer default arguments over overloading if possible. Therefore, you follow the DRY principle (don't repeat yourself).

```
void print(const string& s, format f = {});
```

The equivalent functionality with overloading requires two functions:

```
void print(const string& s); // use default format  
void print(const string& s, format f);
```

F.55

Don't use `va_arg` arguments

The title of this rule is too short. Use variadic templates instead of `va_arg` arguments when your function should accept an arbitrary number of arguments.

Variadic functions are functions such as `std::printf` that can take an arbitrary number of arguments. The issue is that you have to assume that the correct types were passed. Of course, this assumption is very error prone and relies on the discipline of the programmer.

To understand the implicit danger of variadic functions, here is a small example.

```
// vararg.cpp

#include <iostream>
#include <cstdarg>

int sum(int num, ... ) {

    int sum = 0;

    va_list argPointer;
    va_start(argPointer, num );
    for( int i = 0; i < num; i++ )
        sum += va_arg(argPointer, int );
    va_end(argPointer);

    return sum;
}

int main() {

    std::cout << "sum(1, 5): " << sum(1, 5) << '\n';
    std::cout << "sum(3, 1, 2, 3): " << sum(3, 1, 2, 3) << '\n';
    std::cout << "sum(3, 1, 2, 3, 4): "
        << sum(3, 1, 2, 3, 4) << '\n'; // (1)
    std::cout << "sum(3, 1, 2, 3.5): "
        << sum(3, 1, 2, 3.5) << '\n'; // (2)

}
```

sum is a variadic function. Its first argument is the number of arguments that should be summed up. The following background information about `va_arg` macros helps with understanding the code.

- **va_list**: holds the necessary information for the following macros
- **va_start**: enables access to the variadic function arguments
- **va_arg**: accesses the next variadic function argument
- **va_end**: ends the access of the variadic function arguments

For more information, read cpreference.com about variadic functions.

In (1) and (2), I had a bad day. First, the number of the arguments `num` is wrong; second, I provided a `double` instead of an `int`. The output shows both issues. The last element in (1) is missing, and the `double` is interpreted as `int` (2). See Figure 4.7.

```
rainer : bash — Konsole
File Edit View Bookmarks >
rainer@seminar:~> vararg
sum(1, 5): 5
sum(3, 1, 2, 3): 6
sum(3, 1, 2, 3, 4): 6
sum(3, 1, 2, 3.5): 539767595
rainer@seminar:~> █
rainer : bash
```

Figure 4.7 *Summation with `va_arg`*

These issues can be easily overcome with fold expressions in C++17. In contrast to `va_args`, fold expressions automatically deduce the number and the type of their arguments.

```
// foldExpressions.cpp

#include <iostream>

template<class ... Args>
auto sum(Args ... args) {
    return (... + args);
}

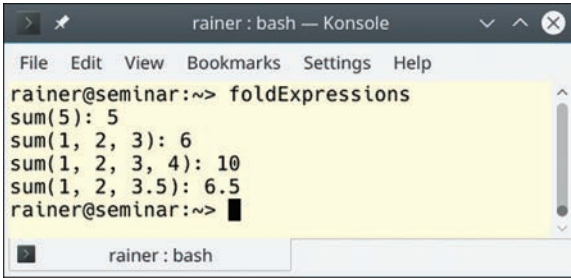
int main() {

    std::cout << "sum(5): " << sum(5) << '\n';
    std::cout << "sum(1, 2, 3): " << sum(1, 2, 3) << '\n';
    std::cout << "sum(1, 2, 3, 4): " << sum(1, 2, 3, 4) << '\n';
    std::cout << "sum(1, 2, 3.5): " << sum(1, 2, 3.5) << '\n';

}
```

The function `sum` may look scary to you. It requires at least one argument and uses C++11 variadic templates. These are templates that can accept an arbitrary number of arguments. The arbitrary number is held by a so-called parameter pack denoted by an ellipsis (`...`). Additionally, with C++17, you can directly reduce a parameter pack with a binary operator. This addition, based on variadic templates, is called fold expressions. In the case of the `sum` function, the binary `+` operator (`... + args`) is applied. If you want to know more about fold expressions in C++17, details are at <https://www.modernescpp.com/index.php/fold-expressions>.

The output of the program is as expected (see Figure 4.8).



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> foldExpressions
sum(5): 5
sum(1, 2, 3): 6
sum(1, 2, 3, 4): 10
sum(1, 2, 3.5): 6.5
rainer@seminar:~> █
```

Figure 4.8 *Summation with fold expressions*

Related rules

An additional rule to lambdas is in Chapter 8, Expressions and Statements: “ES.28: Use lambdas for complex initialization, especially of const variables.”

I skipped the C++20 feature `std::span` in this chapter and provided basic information on `std::span` in Chapter 7, Resource Management.

Distilled

Important

- A function should perform one operation, be short and simple, and have a carefully chosen name.
- Make functions that could run at compile-time `constexpr`.
- Make your functions pure if possible.
- Distinguish between the in, in/out, and out parameters of a function. Use passing by value or by const reference for in, use passing by reference for in/out, and use passing by value for the out parameter.
- Passing parameters to functions is a question of ownership semantics. Passing by value makes the function an independent owner of the resource. Passing by pointer or reference means the function only borrows the resource. A `std::unique_ptr` transfers the ownership to the function. `std::shared_ptr` makes the function a shared owner.
- Use variadic templates instead of `va_arg` arguments when your function should accept an arbitrary number of arguments.

Index

Symbols

- () (parentheses), 187
- { } (curly braces), 144, 166

A

- ABI (application binary interface), 23–24
- Abrahams, David, 280
- abstract class, 101, 102
- abstraction, 167, 302–304
- access
 - memory, 225–229
 - nonexisting element of a `std::string`, 404
 - objects, 114–117
 - sequence containers, 229
- accounts, comparing, 326
- accumulate algorithm, 166
- acquire-release semantics, singletons, 218
- ADL (argument-dependent lookup), 126, 314, 315–316
- algorithms, 8
 - Euclidean, 223
 - expressing, 304
 - function objects, 305–307
 - `gcd`, 223–225
 - generic programming, 301, 302. *See also* generic programming
 - parallel, 266
 - preference over raw loops, 201
 - `std::accumulate`, 166
 - `std::transform_exclusive_scan`, 269
 - STL (Standard Template Library), 12, 266
- aliases
 - defining, 311
 - smart pointers, 162–164
 - templates, 310, 311
- ALL_CAPS, 134, 170–171
- allocation
 - memory, 147, 246
 - resource management, 145–150
- analysis, enabling code, 448
- annotated graphs, 243
- anonymous unions, 128–129
- application binary interface. *See* ABI (application binary interface)
- architecture, 423–425
 - code stability, 423
 - cycles among libraries, 425
 - expressing reusable parts as libraries, 424–425
- argument-dependent lookup. *See* ADL (argument-dependent lookup)
- arguments
 - binary callables, 21, 22
 - defaults, 49, 113–114
 - functions, 359
 - metafunctions, 343
 - order of evaluation, 195–196
 - Regular type/SemiRegular type, 313–314
 - template argument deduction, 313
 - templates, 359
 - `va_arg`, 49–52
- arithmetic
 - errors, 208–210
 - rules, 204
 - signed/unsigned integers, 204–208
- array, 401–403
- arrays, deleting, 194
- artificial scope, 144
- assembler instructions, 30
- assertions, 443
- assignments
 - classes, 59–60, 78–80
 - copy-assignment operator, 221, 222
 - pointers, 117
- auto, applying, 171–172
- automatic management of devices, 435, 449
- automatic memory management, 398. *See also* memory
- automatic type deduction, 179
- availability of source code, 376–377

B

bad functions, example of, 28–29
 Bartosz, Milewski, 276
 base classes, 101
 basic exception safety, 280
 behaviors
 default, 71
 defining, 368
 implementation-defined, 9
 regular types, 58
 shadowing, 111–113
 undefined, 9, 22, 42, 63, 234–235. *See also*
 undefined behaviors
 unspecified, 196
 big six, 59, 60, 61, 85, 130, 222
 binary callables, 21, 22
 binding, late, 114
 bit manipulation, 205
 block scope, 166
 Boost C++ Library, 280
 boundaries, 282, 403–404
 bounds
 errors, 403–404
 safety, 439
 built-in types, 283–285, 294
 byte, 409

C

calculating (at compile time), 339–341
 callables
 definition of, 21, 22
 providing, 305
 C-arrays
 bounds errors, 403–404
 size of, 402
 std::array instead of, 401–403
 std::vector instead of, 398–400
 case-sensitivity, 363
 casts
 avoiding, 197
 naming, 198
 Visual Studio compiler, 197
 catch-clauses, ordering, 288
 catch-fire semantics, 9
 catching exceptions, 285–286. *See also*
 exceptions
 categories
 types, 346–349
 C++ Core Guidelines, enforcing, 447–452

chain operations, 42
 characters
 arbitrary, 44
 character-level input, 411
 owning sequences, 405–406
 std::string_view, 407–408
 termination, 405, 406
 cheap operations, 33
 Clang compiler, 172, 196–197, 219
 clang-tidy tool, 450–452
 classes
 abstract, 101, 102
 accessing, 56
 assignments, 59–60, 78–80
 base, 101
 concrete types, 58–59
 constructors, 59–60, 66–78. *See also*
 constructors
 copying, 69, 78–83
 declarations, 431
 default arguments, 113–114
 default constructors, 68–74
 default operations, 60–66, 88–98
 definitions of, 53
 designing, 102–117
 destructors, 59–60, 83–88
 dynamic-cast, 114–117
 enum, 133–134
 explicit actions, 84
 functions, 55
 hierarchies, 59, 98–117, 331
 implementation, 56
 inheritance, 54
 initializing, 71
 invariant, 55
 moving, 78–83
 non-dependent class template members,
 323–325
 objects, 114–117. *See also* objects
 operators, 117–126. *See also* operators
 overloading, 111
 Pimpl idiom, 23–24
 polymorphic, 81–83
 RAII (Resource Acquisition Is
 Initialization), 140–142
 resources, 84
 special constructors, 76–78
 versus struct, 54
 summary rules, 54–58
 unions, 126–129

- classical enumerations, 131, 132. *See also*
 - enumerations
- cleanup actions, 433–435
- clients, communication, 281
- clone function, 103–105
- code. *See also* performance
 - abstraction, 302–304
 - enabling analysis, 448
 - expressing ideas in, 8
 - expressiveness, 307
 - generic code based on
 - templates, 10
 - messy, 12, 13
 - multi-threaded programs, 232–234
 - null pointers, 192–193
 - optimization, 354–356
 - quality of, 167
 - repetition, 291
 - reusing, 232
 - source. *See* source code
 - stability, 423
 - unknown, 249–250
 - writing, 223
 - wrong assumptions, 214–218
- common names, 169–170
- communication
 - error handling, 281–282
 - functions, 20–22
- comparisons, 325–330
 - type-traits library, 349–351
- compiler errors, 332, 387
- Compiler Explorer, 30, 219
- compilers, default operations, 60–66
- compile time, 338
 - calculating at, 338, 339–341
 - checking, 10, 11
 - gcd algorithms, 223–225
 - type manipulation at, 340–341
- complicated expressions, 186
- composite type categories, 347–348
- concepts, 453–456
- concrete types, 58–59
- concurrency, 17, 231–232, 245
 - data sharing, 257–261
 - general guidelines, 232–245
 - lock-free programming, 273–276
 - locks, 246–250
 - message passing, 269–273
 - parallelism and, 232, 266–269
 - resources, 261–264
 - threads, 250–257
 - validating, 238–245
- conditional execution, 360
- condition variables, 254–257
 - versus* tasks, 272
 - without predicates, 257
- configuring applied rules, 449
- consistency
 - of default operations, 63–66
 - initialization preferences, 76
- const
 - casts, 199
 - correctness, 294
 - defining objects, 297–298
 - member functions, 294–296
- constant expressions, 29–30
 - functions, 357
 - templates, 356–362
 - user-defined types, 357–358
 - variables, 356
- constants, 293–298
 - enumerations, 133
 - initializers, 75
 - introducing, 176
 - magic, 190
 - symbolic, 190
- constexpr, 29–30, 298, 342
 - metaprogramming, 358
- constructors
 - calling virtual functions, 91–98
 - classes, 59–60, 66–78
 - conversion, 73
 - copy, 65
 - default, 68–74
 - defining, 62
 - delegating, 76, 77
 - explicit, 121
 - inheriting, 77–78
 - special, 76–78
 - throwing exceptions, 68
- containers, 23
 - expressing, 305
 - iterating through, 366
 - sequence, 229
 - STL (Standard Template Library), 60, 398–404
 - threads as, 250–251
- context
 - minimizing dependencies, 320–321
 - minimizing switching, 261

contracts, 457–458
 interfaces, 15–16. *See also* interfaces

conventional usage, 118–126

conversions
 constructors, 121
 decay, 117
 expressions, 197–199
 implicit conversion operators, 122–124
 narrowing, 180–182

copy-and-swap idiom, 93–94

copy-assignment operator, 63, 221, 222

copy constructors, 65

copying, 42
 classes, 69, 78–83
 deep copying, 80
 parameters, 34
 semantics, 80–83
 shallow copying, 80

copy-only type, 221

copy semantics, 12

core dumps, 44, 146

correctness, type-traits library, 353–354

covariant return type, 103, 104

.cpp files, 384, 386–387

CppMem, 241–245

C-strings, 406

C-style programming, 375
 availability of source code, 376–377
 entire source code not available, 378–380
 preference for C++, 375–376
 using interfaces for, 378–380

curly braces {}, 144, 166

cycles
 breaking, 154–156
 of smart pointers, 154, 156

cyclic dependencies, 388–390

D

data members
 accessing specifiers, 105
 non-const, 107

data races, 234–235
 in CppMem, 245

data sharing, concurrency, 257–261

deadlocks, 248

deallocation, resource management, 145–150

decay, 117

declaration
 classes, 431
 expressions, 168
 functions, 427–428
 naming, 168–169, 171
 statements, 168
 static_assert, 10

deduction
 automatic type, 179
 template argument, 311–312

deep copying, 80. *See also* copying
 polymorphic classes, 103

=default, 89–90

defaults
 arguments, 49, 113–114
 behaviors, 71
 constructors, 68–74, 69–74
 operations, 60–66, 88–98
 statements, 202–204

=delete, 89, 90–91

deleting
 arrays, 194
 destructors, 91

dependencies
 avoiding, 390
 cyclic, 388–390
 injection, 18–20
 minimizing context, 320–321
 non-dependent class template members,
 323–325
 between special member functions, 61

deque, 226

dereferencing pointers, 191–193, 194

design
 classes, 102–117
 error handling, 281–282
 Gang of Four (GoF), 111
 optimizations, 219–222

Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, Helm, Johnson, Vlissides), 111

destructors, 145
 calling virtual functions, 91–98
 classes, 59–60, 83–88
 defining, 84, 85
 deleting, 91
 failing, 88
 need for, 83

- nonvirtual, 87
- protected, 87
- public, 86–87
- virtual, 86–87
- detection
 - errors, 10
 - overflow, 208
- deterministic, definition of, 437
- devices, automatic management of, 435, 449
- direct ownership, 286–287
- discriminated unions, 126
- documentation, intention of, 9
- don't repeat yourself. *See* DRY (don't repeat yourself)
- do while loops, 199, 200
- DRY (don't repeat yourself), 49, 291, 364, 365
- dynamic-cast, 114–117

E

- enable_if, 319–320, 352, 356
- enabling code analysis, 448
- endl, 415–418
- enforcing rules, 447–452
- enumerations, 131–137
 - ALL_CAPS, 134
 - constants, 133
 - enum class, 133–134
 - enumerator values, 136
 - over macros, 132–133
 - strongly typed enums, 131
 - underlying types, 135
 - unnamed, 134–135
- equality operators, 94–96, 97–98
- equivalent operations, 124–125
- error handling, 279, 280–281. *See also* errors
 - design, 281–282
 - implementation, 283–291
- errors
 - arithmetic, 208–210
 - bounds, 403–404
 - with C++ compilers, 377
 - compilation, 174
 - compiler, 387
 - detecting, 10
 - run-time, 11
 - SFINAE (Substitution Failure Is Not An Error), 320, 352
 - single pointers and, 22–23

- static_assert declarations, 10
- use-before-set, 177
- Euclidean algorithm, 223
- evaluation, order of, 150, 194–195, 194–196
- exceptions
 - avoiding, 429–431
 - basic exception safety, 280
 - catching, 285–286
 - dynamic-cast, 116
 - error handling, 283. *See also* error handling
 - ordering catch-clauses, 288
 - purpose-designed user-defined types, 283–285
 - safety, 280
 - sending, 270–271
 - specifications, 287–288
 - strong exception safety, 280
 - throwing, 68
- execution
 - conditional, 360
 - metafunctions, 342
 - policies, 267
 - selecting, 243
 - single-threaded, 218
- explicit constructors, 121
- explicit sharing, minimizing, 236–237
- expressions, 165–166, 186
 - complicated, 186
 - constant, 29–30
 - conversions, 197–199
 - general rules, 166–168
 - magic constants, 190
 - operator precedence, 187
 - order of evaluation, 194–195
 - pointers, 187–190, 191–193
 - range checking, 190–191
 - statements, 148–150
 - summation with fold, 51
- expressiveness, 46–47
 - function objects, 307

F

- failing destructors, 88
- failure transparency, 280
- fallthrough, 201, 202
- files
 - .cpp, 384, 386–387
 - .h, 384, 385–386, 388

- header, 391, 393–394
 - source. *See* source files
- final, 102, 103
- flushing, 415–418
- fold expressions, summation with, 51
- for loops, 199, 200
- format strings, 413–414
- for-statements, 168–169
- forwarding, perfect, 333–335
- forward_list, 226
- forward parameters, 34–36
- forward std, 144, 198, 226
- free, 145–146
- func, 39
- function objects
 - advantages of, 307–310
 - algorithms, 305–307
 - as sorting criteria, 307
 - with state, 309
- functions, 27–28
 - ADL (argument-dependent lookup), 126
 - arguments, 359
 - callable, 21, 22
 - classes, 55
 - cleanup actions, 433–435
 - clone, 103–105
 - communication, 20–22
 - constant expressions, 357
 - constexpr, 29–30, 298, 342, 356
 - declarations, 427–428
 - =default, 89–90
 - defining, 368, 385
 - definitions, 28–29
 - =delete, 89, 90–91
 - dependencies between special member, 61
 - free, 145–146
 - helper, 57
 - invariants for, 15
 - ISP (interface segregation principle), 100
 - lambdas, 46
 - main, 42
 - malloc, 145–146
 - members, 56
 - metafunctions, 342–345
 - naming, 28–29
 - noexcept, 30–31, 88
 - nonmember, 118–122
 - order of evaluation, 195–196
 - overloading, 379

- parameter packs, 35, 51, 335, 336
- parameters, 156–159
- passing parameters, 32–38. *See also*
 - parameters
- printf, 413–414
- pure, 31–32
- refactoring, 29
- return-statements in, 428–429
 - as sorting criteria, 307
- specialization of templates, 367–372
- std::move(local), 45–46
- struct, 55
- swap, 92–94
- templates, 311–313
- unit tests, 18
- virtual, 82, 102, 223, 331, 357
- virtual clone member, 105

fundamental types, 176, 181

G

- Gang of Four design. *See* GoF (Gang of Four) design
- GCC compiler, 172, 174, 196, 197, 219, 354
- gcd algorithm, 223–225
- general rules. *See* rules
- generic code. *See also* code
 - templates, 10
- generic programming, 301, 302
- getters, 106
- global containers, threads as, 250–251
- global scope, 176
- global variables, non-const, 16–17
- GoF (Gang of Four) design, 111
- graphs, annotated, 243
- GSL (Guidelines Support Library), 441–444
 - assertions, 443
 - ownership pointers, 442–443
 - utilities, 443–444
 - views, 441–442
- guarantees
 - evaluation order, 150, 196
 - exceptions, 280. *See also* exceptions
- Guidelines Support Library. *See* GSL (Guidelines Support Library)

H

- hardware/compiler combinations, 274–276
- Haskell, 32

- header files, 391
 - unnamed namespaces, 394
 - using namespaces, 393–394
- helper functions, 57
- .h files, 384, 385–386
 - #include guard, 388
- hiding
 - implementation details, 310
 - member functions, 174
- hierarchies
 - catching exceptions, 285–286
 - classes, 53, 59, 98–117, 331. *See also* classes
 - equality operators, 97–98
 - navigating, 115
 - templates, 330–332
- hints, compilers, 223

I

- IILE (Immediately Invoked Lambda Expression), 183
- Immediately Invoked Lambda Expression. *See* IILE (Immediately Invoked Lambda Expression)
- immutable data, 12, 293–298
- implementation
 - classes, 56
 - error handling, 283–291
 - hiding details, 310
 - inheritance, 98, 107–111
 - Plmpl idiom, 23–24, 109
 - singletons, 18
 - source files, 384–391
 - templates with specializations, 325–330
 - ThreadSanitizer, 239–241
- implementation-defined behavior, 9, 205
- implicit conversion operators, 122–124
- in-class initializers, 75
- inheritance
 - classes, 54
 - implementation, 98, 107–111
 - interfaces, 107–111
 - multiple, 107, 111
- initialization, 177–182
 - lambdas, 183–184
 - objects, 175–176
 - two-phase, 431–433
 - variables, 175–184

- injection
 - dependency, 18–20
- inline, 30, 354, 357, 385
- in-out parameters, 36
- in parameters, 34
- input/output. *See* I/O (input/output)
- integers, signed/unsigned, 204–208
- interfaces, 15–16
 - ABI (application binary interface), 23–24
 - abstract classes, 101
 - applying classes, 56
 - base classes, 101
 - C-style programming, 378–380
 - defining, 386–387
 - dependency injection, 18–20
 - function communication, 20–22
 - functions, 27–28. *See also* functions
 - inheritance, 107–111
 - multiple inheritance, 111
 - non-const global variables, 16–17
 - segregation principle, 100
 - single pointers, 22–23
 - singletons, 17–18
 - source files, 384–391
 - templates, 305–320
- interface segregation principle. *See* ISP (interface segregation principle)
- internal linkage, 394–395
- invariants
 - error handling, 282
 - for functions, 15
- I/O (input/output)
 - character-level input, 411
 - iostreams, 413–414
 - std::endl, 415–418
 - STL (Standard Template Library), 411–418
 - stream synchronization, 414–415
- iostreams, 411, 413–414
- ISO Standards, 8, 9
- ISP (interface segregation principle), 100
- iteration
 - statements, 199–201
 - through containers, 366
- iterator categories, 366–367

J

- jthread, 251–253

K

keep it simple, stupid. *See* KISS (keep it simple, stupid)
 keywords, overriding, 102
 KISS (keep it simple, stupid), 59
 Koenig lookup, 126, 314, 315–316

L

lambdas, 46
 initializing, 183–184
 overloading, 117
 references, 47, 48–49
 as sorting criteria, 307
 with state, 310
 unnamed, 364–365
 languages
 features, 167–168
 Haskell, 32
 late binding, 114
 leaks
 memory, 287
 resources, 11
 libraries, 166. *See also* STL (Standard Template Library)
 Boost C++ Library, 280
 cycles among, 425
 expressing reusable parts as, 424–425
 GSL (Guidelines Support Library), 441–444
 reusing code, 232
 STL (Standard Template Library). *See* STL (Standard Template Library)
 support, 13
 type-traits, 345–346, 351
 life cycles of objects, 59
 lifetimes
 objects, 142
 safety, 439
 semantics, 157–159
 limiting scope, 168–169
 linker errors, 387
 Linux systems
 creation of threads, 262
 flushing, 418
 size of threads, 261
 list, 226
 literal type, 224
 LoadLoad, 276

local names, 169–170
 local objects, 140, 141, 287. *See also* objects
 lock-free programming, 273–276
 lock_guard, 264–266
 locks
 concurrency, 246–250
 mutexes, 264
 std::lock_guard, 264–266
 std::unique_lock, 264–266
 logical constness, 295
 logical structure, expressing, 394
 lookups, 126, 314, 315
 loops, 199, 200
 raw, 201
 recursion *versus*, 360
 lost wakeups, 255, 256
 lvalue references, 42–46

M

macros
 enumerations over, 132–133
 statements, 184–185
 magic constants, 190
 malloc, 145–146
 management
 automatic management of devices, 435, 449
 automatic memory management, 398
 memory, 398, 405
 resources, 139–140. *See also* resource management
 Martin, Robert C., 100
 mathematical functions, 32. *See also* pure functions
 member functions, 56
 boundary-checking, 403–404
 compiler errors, 332
 const, 294–296
 hiding, 174
 reserve, 399
 shadowing, 112
 virtual member function templates, 331–332
 members
 accessing specifiers for data, 105
 declaring, 74
 dependencies between special functions, 61
 initializing, 74–76, 75
 ISP (interface segregation principle), 100

- minimizing exposure, 58
 - non-dependent class template, 323–325
 - nonpublic, 58
 - parameters, 321–322
 - pointers, 85
 - variables, 75
 - memory
 - accessing, 225–229
 - allocation, 147, 246
 - automatic memory management, 398
 - leaks, 287
 - management, 398, 405
 - models, 242
 - ROM (readonly memory), 29
 - saving, 126
 - sequence containers, 229
 - messages
 - as exceptions and values, 271
 - passing, 269–273
 - messy code, 12, 13
 - metadata, templates, 341
 - metafunctions, templates, 342–345
 - metaprogramming, 336–356
 - constexpr function, 358
 - templates, 351
 - meta-rules. *See* philosophical rules
 - Metaware compiler, 338
 - Meyers singleton, 217
 - Microsoft Visual Studio Compiler, 30
 - minimizing
 - context dependencies, 320–321
 - context switching, 261
 - thread creation/destruction, 261–263
 - time locking mutexes, 264
 - models, memory, 242
 - modification *versus* new value, 359
 - most vexing parse, 177, 179
 - move(local) function, 45–46
 - moving
 - classes, 78–83
 - semantics, 80–83
 - std::unique_ptr, 153
 - MSVC compiler, 172
 - multiple inheritance, 107, 111
 - multiple mutexes, acquiring, 247–248
 - multi-threaded programs, 232–234
 - mutable data, 12
 - mutexes
 - acquiring, 247–248
 - locks, 264
 - MyGuard, 265
 - myths, 427–436
- ## N
- naked unions, 127–128
 - Named Return Value Optimization. *See* NRVO (Named Return Value Optimization)
 - names
 - ALL_CAPS, 170–171
 - casts, 198
 - common names, 169–170
 - conventions, 342
 - declarations, 168–169, 171
 - expressions, 168–185
 - functions, 28–29
 - local names, 169–170
 - mangling, 380
 - operations, 362–364
 - redundancy, 171–172
 - reusing names, 172–175
 - similar names, 170
 - statements, 168–185
 - templates, 314–319
 - namespaces, 57
 - defining overloaded operators, 125–126
 - source code, 391–395
 - narrowing conversion, 180–182
 - negative values, 206–208
 - nested scopes, reusing names, 172–175
 - NNM (No Naked Mutex), 246
 - NNN (No Naked New), 147, 246
 - noexcept, 30, 31, 79, 95
 - destructors, 88
 - function definition, 88
 - noexcept function, 30–31, 88
 - no exception safety, 280
 - no-leak guarantee, 280
 - No Naked Mutex. *See* NNM (No Naked Mutex)
 - No Naked New. *See* NNN (No Naked New)
 - non-const data members, 107
 - non-const global variables, 16–17
 - non-dependent class template members, 323–325
 - nongeneric code, writing, 365–367
 - nonmember functions, 118–122
 - nonpublic members, 58
 - nonrules, 427–436

nonvirtual, destructors, 87
 normal parameter passing, 32
 no-throw guarantees, 280
 notifications
 sending, 272–273
 with tasks, 273
 NRVO (Named Return Value Optimization),
 36, 37
 NULL, 192
 nullptr, 191–193, 192–193. *See also* pointers

O

objects

- accessing, 114–117
- constructors creating, 67
- creating, 145
- defining, 297–298, 385–386
- direct ownership, 286–287
- function objects. *See* function objects
- immutable data and, 294
- initializing, 175–176
- life cycles of, 59
- lifetimes, 142
- local, 140, 141
- moving, 80
- scoped, 143–144

ODR (One Definition Rule), 385

One Definition Rule. *See* ODR (One
 Definition Rule)

operands, defining overloaded operators,
 125–126

operations

- cheap, 33
- equivalent, 124–125
- naming, 362–364
- naming functions, 28–29
- passing to algorithms, 305–307
- reordering, 276

operators

- ADL (argument-dependent lookup), 126
- conventional usage, 118–126
- copy-assignment, 221, 222
- defining, 125–126
- equality, 94–96, 97–98
- implicit conversion, 122–124
- overloading, 117–126
- precedence, 187
- symmetric, 118–122

optimization, 17

- code, 167
- design, 219–222
- enabling, 218–229
- type-traits library, 354–356
- wrong, 214

order of evaluation, expressions, 194–195

out_of_range, 190, 403

out parameter, 52

output parameters, 36–37

out values, 37–38

overflow, 208

overloading, 49

- ADL (argument-dependent lookup), 126
- classes, 111
- conventional usage, 118–126
- defining operators, 125–126
- functions, 379
- function templates, 369–370
- implicit conversion operators, 122–124
- operators, 117–126

override, 102

ownership

- direct, 286–287
- pointers/references, 143, 442–443
- semantics, 38–41
- sharing, 164, 258–261
- std::shared_ptr, 151–153
- std::unique_ptr, 150–151

P

packing parameters, 35

parallelism, concurrency and, 232, 266–269

parameters

- in, 34
- forward, 34–36
- functions, 156–159
- in-out, 36
- members, 321–322
- normal parameter passing, 32
- out, 52
- output, 36–37
- ownership semantics, 38–41
- packs, 35, 51, 335, 336
- passing, 32–38
- value return semantics, 42–46

Parent, Sean, 167

parentheses (), 187

- passing
 - messages, 269–273
 - pointers, 297
 - references, 297
 - passing parameters, 32–38
 - normal parameter passing, 32
 - ownership semantics, 38–41
 - value return semantics, 42–46
 - perfect forwarding, 333–335
 - performance, 213
 - enabling optimization, 218–229
 - function objects, 307
 - measurements, 214
 - Meyers singleton, 217
 - wrong assumptions, 214–218
 - wrong optimizations, 214
 - philosophical rules, 7
 - compile-time checking, 10
 - expressing ideas in code, 8
 - expressing intent, 9
 - immutable data, 12
 - messy code, 12, 13
 - resource leaks, 11
 - run-time checking, 11
 - run-time errors, 11
 - saving space and time, 11–12
 - statically type safe programs, 10
 - supporting tools, 13
 - support libraries, 13
 - writing in ISO standard C++, 8–9
 - physical constness, 295
 - Pikus, Fedor, 274
 - Plain Old Data. *See* POD (Plain Old Data)
 - Plmpl idiom, 23–24, 109
 - POD (Plain Old Data), 33
 - pointers, 42–46, 84
 - assigning, 117
 - dereferencing, 191–193, 194
 - dynamic-cast, 114, 115, 116
 - expressions, 187–190, 191–193
 - members, 85
 - null, 191–193
 - passing, 162–164, 297
 - Plmpl idiom, 23–24, 109
 - raw, 85, 140, 143
 - rules, 191
 - single, 22–23
 - smart, 150–164
 - policy execution, 266, 267
 - polymorphic classes, 81–83, 103
 - portably enforceable, definition of, 437
 - POSIX Threads, 256
 - postconditions, 15
 - power as function/metafunction, 345
 - pragma once, 388
 - preconditions, 15
 - predicates, condition variables without, 257
 - predictability, 225–229
 - Preshing, Jeff, 276
 - primary type categories, 346–349
 - principle of least astonishment, 118
 - printf function, 413–414
 - profiles, 437–439
 - Pro.boundsBounds safety, 439
 - Pro.lifetimeLifetime safety, 439
 - Pro.typeType safety, 438
 - programming
 - C-style, 375. *See also* C-style programming
 - generic. *See* generic programming
 - metaprogramming, 336–356, 351, 358. *See also* metaprogramming
 - programs
 - multi-threaded, 232–234
 - statically type safe, 10
 - property types, 348–349
 - protected data, 106–107, 436
 - protected destructors, 87
 - public destructors, 86–87
 - pure functions, 31–32. *See also* functions
 - purpose-designed user-defined types, 283–285
- ## Q
- quality of code, 167
- ## R
- race conditions, 253
 - RAII (Resource Acquisition Is Initialization), 140–142, 246, 264, 287, 288, 289
 - range checking, 190–191
 - ranges, expressing, 305
 - raw loops, 201
 - raw pointers, 85, 140, 143
 - raw references, 143
 - read-only memory. *See* ROM (read-only memory)
 - recursion *versus* loop, 360
 - redundancy, naming, 171–172
 - refactoring, 17, 29

- references, 84, 140
 - catching exceptions, 285–286
 - dynamic-cast, 114, 115, 116
 - lambdas, 47, 48–49
 - to locals, 42–44
 - lvalue, 42–46
 - passing, 297
 - raw, 143
 - SemiRegular type, 315
 - universal, 334
- referential transparency, 31
- Regular, 313–314
- regular types, 58, 59
- relations, enabling, 243
- relaxed semantics, 275
- reordering operations, 276
- repetition of code, 291
- reserve function, 399
- Resource Acquisition Is Initialization.
 - See* RAII (Resource Acquisition Is Initialization)
- resource management, 139–140
 - allocation, 145–150
 - deallocation, 145–150
 - general rules, 140–144
 - smart pointers, 150–164
- resources
 - concurrency, 261–264
 - leaks, 11
 - ownership of, 39
- return-statements in functions, 428–429
- Return Value Optimization. *See* RVO (Return Value Optimization)
- return values
 - metafunctions, 343
 - simulations, 361
- reusing
 - expressing reusable parts as libraries, 424–425
 - operations, 362–364
- ROM (read-only memory), 29
- rule of five, 61, 83, 84, 89
- rule of six, 61
- rule of zero, 60
- rules
 - arithmetic, 204
 - class hierarchies, 99–101
 - configuring applied, 449
 - enforcing, 447–452

- expressions, 166–168
- interfaces, 20–22
- NNN (No Naked New), 147
- ODR (One Definition Rule), 385
- passing parameters, 32–38
- philosophical, 7. *See also* philosophical rules
- pointers, 191
- resource management, 140–144
- statements, 166–168
- summary, 54–58
- templates, 362–372
- user-defined types, 53
- running code analysis, 449
- run time
 - calculating at, 343
 - checking, 10, 11
 - constant expression in ROM, 29–30
 - errors, 11
 - gcd algorithms, 223–225
- RVO (Return Value Optimization), 36, 37

S

- safety
 - basic exception, 280
 - bounds, 439
 - exceptions, 280
 - lifetimes, 439
 - types, 438
- scoped enums, 131, 135, 137
- scoped objects, 143–144
- scopes
 - block, 166
 - global, 176
 - limiting, 168–169
 - reusing names, 172–175
 - sizes, 168
- selection statements, 201–204
- self-assignment, 79–80
- semantics
 - acquire-release, 218
 - catch-fire, 9
 - copy, 65
 - copying, 80–83
 - copy-only type, 221
 - lifetime, 157–159
 - moving, 80–83
 - ownership, 38–41

- relaxed, 275
 - summary rules, 54–58
 - value return, 42–46
- SemiRegular, 313–314
- sequence containers, 229
- sequential consistency, 218, 243, 274, 275
- setters, 106
- SFINAE (Substitution Failure Is Not An Error), 320, 352
- shadowing, 111–113
- shallow copying, 80. *See also* copying
- sharing ownership, 258–261
- signed/unsigned integers, 204–208
- SIMD (Single Instruction, Multiple Data), 267
- similar names, 170. *See also* naming
- simulations, return values, 361
- single-argument constructors, 72–74
- Single Instruction, Multiple Data. *See* SIMD (Single Instruction, Multiple Data)
- single pointers, 22–23
- single return-statements in functions, 428–429
- single-threaded case, 218
- singletons, 17–18
- six, rule of, 61
- sizes
 - of C-arrays, 402
 - of chars with C++ compilers, 377
 - of enumerators, 136
 - of scopes, 168
 - of threads, 261
 - of vectors, 399, 400
- slicing, 81
- smart pointers, 150–164
 - aliases, 162–164
 - cycles of, 154
 - as function parameters, 156
 - lifetime semantics of, 157–159
 - sharing ownership, 261
 - `std::unique_ptr`, 160–162
- software units, 281
- source code
 - availability of, 376–377
 - entire code not available, 378–380
 - namespaces, 391–395
- source files, 383, 384
 - cyclic dependencies, 388–390
 - implementation, 384–391
 - interfaces, 384–391
- span, 10, 23, 52, 101
- special constructors, 76–78
- specialization,
 - function templates, 367–372
 - templates, 360
- special member functions, 61
- specifications, exceptions, 287–288
- spurious wakeups, 256
- stability, code, 423
- Standard Template Library. *See* STL (Standard Template Library)
- state, function objects, 308–310
- statements, 165–166, 199. *See also* declarations
 - defaults, 202–204
 - definitions of, 166
 - expressions, 148–150
 - for-statements, 168–169
 - general rules, 166–168
 - initializing variables, 175–184
 - iteration, 199–201
 - macros, 184–185
 - naming, 168–185
 - return-statements in functions, 428–429
 - selection, 201–204
 - switch, 201–202
- statically type safe programs, 10
- `static_assert` declarations, 10
- static type systems, 222–223
- `std::forward`, 144, 198, 226, 334
- `std::make_unique`, 36, 148, 153, 334, 335
- `std::shared_ptr`, 140, 151–153
- `std::unique_ptr`, 140, 146–147, 150–151
 - moving, 152
 - smart pointers, 160–162
- `std::weak_ptr`, 140, 154–156
- STL (Standard Template Library), 8, 21, 305, 397
 - algorithms, 8, 12, 266
 - containers, 23, 60, 398–404
 - expressions, 166
 - I/O (input/output), 411–418
 - RAII (Resource Acquisition Is Initialization), 140–142
 - strings, 410
 - text, 404–411
- streams
 - state, 411–413
 - synchronization, 414–415
- strings
 - accessing nonexistent element of, 404
 - format, 413–414

- owning character sequences, 405–406
- STL (Standard Template Library), 410
- string_view, referring to character sequences, 407–408
- strong exception safety, 280
- strongly typed enums, 131
- Stroustrup, Bjarne, 169
- struct, 55
 - case-sensitivity, 363
 - class *versus*, 54
- structures, organizing data into, 54–55
- Substitution Failure Is Not An Error. *See* SFINAE (Substitution Failure Is Not An Error)
- suffixes, .cpp, 384
- summary rules, 54–58
- summation
 - with fold expressions, 51
 - with va_arg, 51
- support
 - C-arrays, 403–404
 - libraries, 13
 - tools, 13
- Sutter, Herb, 274, 276
- swap function, 92–94
- switch statements, 201–202
- symbolic constants, 190
- symmetric operators, 118–122
- synchronization, 255
 - streams, 414–415
 - volatile for, 238
- sync_with_stdio, 404, 415

T

- tagged unions, 128–129
- tasks
 - condition variables *versus*, 272
 - notifications with, 273
 - versus* threads, 237–238
- Technical Report on C++ Performance*, 430
- templates, 301, 302
 - aliases, 310, 311
 - applying, 302–305
 - argument deduction, 313
 - arguments, 359
 - constant expressions, 356–362
 - defining, 320–330
 - faking concepts, 319–320
 - function objects. *See* function objects
 - functions, 311–313
 - function template specialization, 367–372
 - generic code, 10
 - hierarchies, 330–332
 - implementations with specializations, 325–330
 - instantiation, 338
 - interfaces, 305–320
 - metadata, 341
 - metafunctions, 342–345
 - metaprogramming, 336–356, 351. *See also* metaprogramming
 - naming, 314–319
 - parameter packs, 35, 51, 335, 336
 - Regular type, 313–314
 - rules, 362–372
 - specialization, 360
 - STL (Standard Template Library), 8
 - variadic, 332–336
 - virtual member function, 331–332
- terminate, 88, 251, 252, 287
- termination characters, 405, 406
- testability, 16
- text
 - STL (Standard Template Library), 404–411
 - types of, 405
- threads
 - concurrency, 250–257
 - creation/destruction, 261–263
 - detaching, 253
 - as global containers, 250–251
 - joining, 250
 - passing data to, 257–258
 - POSIX Threads, 256
 - sharing ownership, 258–261
 - sizes of, 261
 - std::jthread, 251–253
 - versus* tasks, 237–238
- ThreadSanitizer, 239–241
- throwing exceptions, 68
 - direct ownership, 286–287
 - troubleshooting, 288–291
- throwing functions, 30–31. *See also* functions
- tools, 238
 - clang-tidy, 450–452
 - CppMem, 241–245

- supporting, 13
- ThreadSanitizer, 239–241
- transform_exclusive_scan algorithm, 269
- transform_reduce, 21, 22
- transparency, referential, 31
- troubleshooting throwing exceptions, 288–291
- two-phase initializations, 431–433
- typedef, defining aliases, 311
- types
 - automatic type deduction, 179
 - built-in, 283–285, 294
 - categories, 346–349
 - concrete, 58–59
 - copy-only, 221
 - fundamental, 176, 181
 - literal, 224
 - manipulation at compile time, 340–341
 - modifying, 351–352
 - properties, 348–349
 - purpose-designed user-defined, 283–285
 - regular, 58, 59
 - Regular, 313–314
 - return, 103, 104
 - safety, 438
 - SemiRegular, 313–314
 - static type systems, 222–223
 - of text, 405
 - underlying, 135
 - unsigned, 205
 - user-defined, 357–358
- type-traits library, 345–346
 - comparisons, 349–351
 - correctness, 353–354
 - metaprogramming, 351
 - modifying types, 351–352
 - optimization, 354–356
 - type categories, 346–349

U

- Uncle Bob, 100
- undefined behaviors, 9, 22, 42, 63
 - core dumps, 146
 - C-strings, 405
 - data races, 234–235
 - naked unions, 127
 - order of evaluation, 194–195
 - printf function, 414

- underflow, 208. *See also* overflow
- underlying types, 135
- unions, 126–129
 - anonymous, 128–129
 - discriminated, 126
 - naked, 127
 - saving memory, 126–128
 - tagged, 128–129
- unique_lock, 264–266
- unit tests, 18
- universal references, 334
- unknown code, calling, 249–250
- unnamed enumerations, 134–135
- unnamed lambdas, 364–365
- unnamed namespaces, 394
- unpacking parameters, 35
- Unruh, Erwin, 337, 338
- unsigned/signed integers, 204–208
- unspecified behavior, 196
- use-before-set error, 177
- user-defined types
 - constant expressions, 357–358
 - rules, 53
- using
 - defining aliases, 311
 - namespaces, 393–394
- utilities. *See* libraries; tools

V

- va_arg arguments, 49–52
- value return semantics, 42–46
- values
 - declaring variables, 176–177
 - enumerations. *See* enumerations
 - enumerator, 136
 - negative, 206–208
 - out, 37–38
 - return, 343
 - sending, 270–271
- Van Eerd, Tony, 274
- variables
 - categories of, 235
 - condition, 254–257. *See also* condition variables
 - constant expressions, 356
 - declaring, 57
 - global, 16–17. *See also* global variables
 - initializing, 175–184, 176

- introducing, 176
- member, 74
- mutable, 296
- naming, 169–170
- purposes of, 182–183
- variadic templates, 332–336
- vector, 398–400, 402–403
- vectorization, 267
- vectors, size of, 399, 400
- views, 441–442
- virtual, 82, 223, 331, 357
- virtual clone member function, 105
- virtual destructors, 86–87
- virtual functions, 102
 - calling, 91–98
 - clone, 105
 - default arguments, 113–114
 - reasons for, 105
- virtuality, 102–105
- virtual member function templates, 331–332
- visibility, modifying, 175
- Visual Studio
 - casts, 197
 - enforcing C++ Core Guidelines, 448–450
- volatile for synchronization, 238

W

- wakeups
 - lost, 255
 - spurious, 256
- warnings with C compilers, 376
- while loops, 199, 200
- Williams, Anthony, 274, 276
- Windows systems
 - creation of threads, 262
 - flushing, 418
 - size of threads, 261
- writing
 - code, 223
 - in ISO standard C++, 8–9
 - nongeneric code, 365–367
- wrong assumptions, 214–218
- wrong optimizations, 214

Y

- YAGNI (you aren't gonna need it), 424

Z

- The Zen of Python, 198, 364, 365
- zero. *See also* arithmetic
 - dividing by, 210
 - rule of, 60