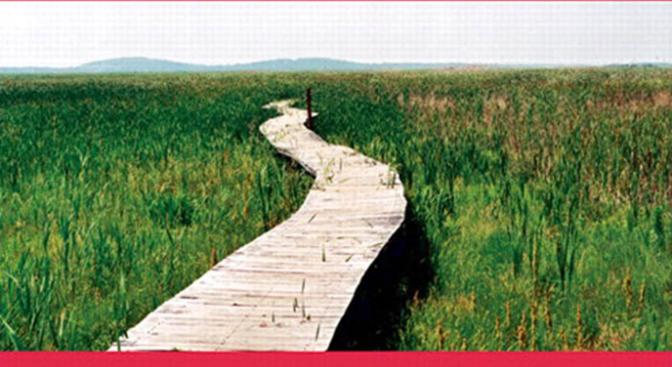
C++ Coding Standards

101 Rules, Guidelines, and Best Practices

Herb Sutter
Andrei Alexandrescu



C++ In-Depth Series • Bjarne Stroustrup

The authors 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.

Publisher: John Wait
Editor in Chief: Don O'Hagan
Acquisitions Editor: Peter Gordon
Editorial Assistant: Kim Boedigheimer
Marketing Manager: Chanda Leary-Coutu
Cover Designer: Chuti Prasertsith
Managing Editor: John Fuller
Project Editor: Lara Wysong
Copy Editor: Kelli Brooks
Manufacturing Buyer: Carol Melville

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the U. S., please contact:

International Sales
international@pearsoned.com

Visit us on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data:

Sutter, Herb.
C++ coding standards : 101 rules, guidelines, and best practices / Herb Sutter, Andrei Alexandrescu.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-11358-6 (pbk. : alk. paper)
C++ (Computer program language) I. Alexandrescu, Andrei. II. Title.

QA76.73.C153S85 2004
005.13'3—dc22

2004022605

Copyright © 2005 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. 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, write to:

Pearson Education, Inc.
Rights and Contracts Department
One Lake Street
Upper Saddle River, NJ 07458

ISBN 0-321-11358-6

Text printed in the United States on recycled paper at Courier Stoughton in Stoughton, Massachusetts.

9th Printing August 2010

Contents

Preface xi

Organizational and Policy Issues 1

0. Don't sweat the small stuff. (Or: Know what not to standardize.) 2
1. Compile cleanly at high warning levels. 4
2. Use an automated build system. 7
3. Use a version control system. 8
4. Invest in code reviews. 9

Design Style 11

5. Give one entity one cohesive responsibility. 12
6. Correctness, simplicity, and clarity come first. 13
7. Know when and how to code for scalability. 14
8. Don't optimize prematurely. 16
9. Don't pessimize prematurely. 18
10. Minimize global and shared data. 19
11. Hide information. 20
12. Know when and how to code for concurrency. 21
13. Ensure resources are owned by objects. Use explicit RAII and smart pointers. 24

Coding Style 27

14. Prefer compile- and link-time errors to run-time errors. 28
15. Use **const** proactively. 30
16. Avoid macros. 32

17. Avoid magic numbers. 34
18. Declare variables as locally as possible. 35
19. Always initialize variables. 36
20. Avoid long functions. Avoid deep nesting. 38
21. Avoid initialization dependencies across compilation units. 39
22. Minimize definitional dependencies. Avoid cyclic dependencies. 40
23. Make header files self-sufficient. 42
24. Always write internal **#include** guards. Never write external **#include** guards. 43

Functions and Operators 45

25. Take parameters appropriately by value, (smart) pointer, or reference. 46
26. Preserve natural semantics for overloaded operators. 47
27. Prefer the canonical forms of arithmetic and assignment operators. 48
28. Prefer the canonical form of **++** and **--**. Prefer calling the prefix forms. 50
29. Consider overloading to avoid implicit type conversions. 51
30. Avoid overloading **&&**, **||**, or **,** (comma) . 52
31. Don't write code that depends on the order of evaluation of function arguments. 54

Class Design and Inheritance 55

32. Be clear what kind of class you're writing. 56
33. Prefer minimal classes to monolithic classes. 57
34. Prefer composition to inheritance. 58
35. Avoid inheriting from classes that were not designed to be base classes. 60
36. Prefer providing abstract interfaces. 62
37. Public inheritance is substitutability. Inherit, not to reuse, but to be reused. 64
38. Practice safe overriding. 66
39. Consider making virtual functions nonpublic, and public functions nonvirtual. 68
40. Avoid providing implicit conversions. 70
41. Make data members private, except in behaviorless aggregates (C-style **struct**s). 72
42. Don't give away your internals. 74
43. Pimpl judiciously. 76
44. Prefer writing nonmember nonfriend functions. 79
45. Always provide **new** and **delete** together. 80
46. If you provide any class-specific **new**, provide all of the standard forms (plain, in-place, and **nothrow**). 82

Construction, Destruction, and Copying **85**
47. Define and initialize member variables in the same order. 86
48. Prefer initialization to assignment in constructors. 87
49. Avoid calling virtual functions in constructors and destructors. 88
50. Make base class destructors public and virtual, or protected and nonvirtual. 90
51. Destructors, deallocation, and **swap** never fail. 92
52. Copy and destroy consistently. 94
53. Explicitly enable or disable copying. 95
54. Avoid slicing. Consider **Clone** instead of copying in base classes. 96
55. Prefer the canonical form of assignment. 99
56. Whenever it makes sense, provide a no-fail **swap** (and provide it correctly). 100

Namespaces and Modules **103**
57. Keep a type and its nonmember function interface in the same namespace. 104
58. Keep types and functions in separate namespaces unless they're specifically intended to work together. 106
59. Don't write namespace **using**s in a header file or before an **#include**. 108
60. Avoid allocating and deallocating memory in different modules. 111
61. Don't define entities with linkage in a header file. 112
62. Don't allow exceptions to propagate across module boundaries. 114
63. Use sufficiently portable types in a module's interface. 116

Templates and Genericity **119**
64. Blend static and dynamic polymorphism judiciously. 120
65. Customize intentionally and explicitly. 122
66. Don't specialize function templates. 126
67. Don't write unintentionally nongeneric code. 128

Error Handling and Exceptions **129**
68. Assert liberally to document internal assumptions and invariants. 130
69. Establish a rational error handling policy, and follow it strictly. 132
70. Distinguish between errors and non-errors. 134
71. Design and write error-safe code. 137
72. Prefer to use exceptions to report errors. 140
73. Throw by value, catch by reference. 144
74. Report, handle, and translate errors appropriately. 145
75. Avoid exception specifications. 146

STL: Containers **149**
76. Use **vector** by default. Otherwise, choose an appropriate container. 150
77. Use **vector** and **string** instead of arrays. 152
78. Use **vector** (and **string::c_str**) to exchange data with non-C++ APIs. 153
79. Store only values and smart pointers in containers. 154
80. Prefer **push_back** to other ways of expanding a sequence. 155
81. Prefer range operations to single-element operations. 156
82. Use the accepted idioms to really shrink capacity and really erase elements. 157

STL: Algorithms **159**
83. Use a checked STL implementation. 160
84. Prefer algorithm calls to handwritten loops. 162
85. Use the right STL search algorithm. 165
86. Use the right STL sort algorithm. 166
87. Make predicates pure functions. 168
88. Prefer function objects over functions as algorithm and comparer arguments. 170
89. Write function objects correctly. 172

Type Safety **173**
90. Avoid type switching; prefer polymorphism. 174
91. Rely on types, not on representations. 176
92. Avoid using **reinterpret_cast**. 177
93. Avoid using **static_cast** on pointers. 178
94. Avoid casting away **const**. 179
95. Don't use C-style casts. 180
96. Don't **memcpy** or **memcmp** non-PODs. 182
97. Don't use unions to reinterpret representation. 183
98. Don't use varargs (ellipsis). 184
99. Don't use invalid objects. Don't use unsafe functions. 185
100. Don't treat arrays polymorphically. 186

Bibliography **187**

Summary of Summaries **195**

Index **209**

Preface

Get into a rut early: Do the same process the same way. Accumulate idioms. ***Standardize.*** *The only difference(!) between Shakespeare and you was the size of his idiom list—not the size of his vocabulary.*

— Alan Perlis [emphasis ours]

The best thing about standards is that there are so many to choose from.

— Variously attributed

We want to provide this book as a basis for your team's coding standards for two principal reasons:

- *A coding standard should reflect the community's best tried-and-true experience:* It should contain proven idioms based on experience and solid understanding of the language. In particular, a coding standard should be based firmly on the extensive and rich software development literature, bringing together rules, guidelines, and best practices that would otherwise be left scattered throughout many sources.
- *Nature abhors a vacuum:* If you don't consciously set out reasonable rules, usually someone else will try to push their own set of pet rules instead. A coding standard made that way usually has all of the least desirable properties of a coding standard; for example, many such standards try to enforce a minimalistic C-style use of C++.

Many bad coding standards have been set by people who don't understand the language well, don't understand software development well, or try to legislate too much. A bad coding standard quickly loses credibility and at best even its valid guidelines are liable to be ignored by disenchanted programmers who dislike or disagree with its poorer guidelines. That's "at best"—at worst, a bad standard might actually be enforced.

How to Use This Book

Think. Do follow good guidelines conscientiously; but don't follow them blindly. In this book's Items, note the Exceptions clarifying the less common situations where the guidance may not apply. No set of guidelines, however good (and we think these ones are), should try to be a substitute for thinking.

Each development team is responsible for setting its own standards, and for setting them responsibly. That includes your team. If you are a team lead, involve your team members in setting the team's standards; people are more likely to follow standards they view as their own than they are to follow a bunch of rules they feel are being thrust upon them.

This book is designed to be used as a basis for, and to be included by reference in, your team's coding standards. It is not intended to be the Last Word in coding standards, because your team will have additional guidelines appropriate to your particular group or task, and you should feel free to add those to these Items. But we hope that this book will save you some of the work of (re)developing your own, by documenting and referencing widely-accepted and authoritative practices that apply nearly universally (with Exceptions as noted), and so help increase the quality and consistency of the coding standards you use.

Have your team read these guidelines with their rationales (i.e., the whole book, and selected Items' References to other books and papers as needed), and decide if there are any that your team simply can't live with (e.g., because of some situation unique to your project). Then commit to the rest. Once adopted, the team's coding standards should not be violated except after consulting with the whole team.

Finally, periodically review your guidelines as a team to include practical experience and feedback from real use.

Coding Standards and You

Good coding standards can offer many interrelated advantages:

- *Improved code quality:* Encouraging developers to do the right things in a consistent way directly works to improve software quality and maintainability.
- *Improved development speed:* Developers don't need to always make decisions starting from first principles.
- *Better teamwork:* They help reduce needless debates on inconsequential issues and make it easier for teammates to read and maintain each other's code.
- *Uniformity in the right dimension:* This frees developers to be creative in directions that matter.

Under stress and time pressure, people do what they've been trained to do. They fall back on habit. That's why ER units in hospitals employ experienced, trained personnel; even knowledgeable beginners would panic.

As software developers, we routinely face enormous pressure to deliver tomorrow's software yesterday. Under schedule pressure, we do what we are trained to do and are used to doing. Sloppy programmers who in normal times don't know good practices of software engineering (or aren't used to applying them) will write even sloppier and buggier code when pressure is on. Conversely, programmers who form good habits and practice them regularly will keep themselves organized and deliver quality code, fast.

The coding standards introduced by this book are a collection of guidelines for writing high-quality C++ code. They are the distilled conclusions of a rich collective experience of the C++ community. Much of this body of knowledge has only been available in bits and pieces spread throughout books, or as word-of-mouth wisdom. This book's intent is to collect that knowledge into a collection of rules that is terse, justified, and easy to understand and follow.

Of course, one can write bad code even with the best coding standards. The same is true of any language, process, or methodology. A good set of coding standards fosters good habits and discipline that transcend mere rules. That foundation, once acquired, opens the door to higher levels. There's no shortcut; you have to develop vocabulary and grammar before writing poetry. We just hope to make that easier.

We address this book to C++ programmers of all levels:

If you are an apprentice programmer, we hope you will find the rules and their rationale helpful in understanding what styles and idioms C++ supports most naturally. We provide a concise rationale and discussion for each rule and guideline to encourage you to rely on understanding, not just rote memorization.

For the intermediate or advanced programmer, we have worked hard to provide a detailed list of precise references for each rule. This way, you can do further research into the rule's roots in C++'s type system, grammar, and object model.

At any rate, it is very likely that you work in a team on a complex project. Here is where coding standards really pay off—you can use them to bring the team to a common level and provide a basis for code reviews.

About This Book

We have set out the following design goals for this book:

- *Short is better than long:* Huge coding standards tend to be ignored; short ones get read and used. Long Items tend to be skimmed; short ones get read and used.

- *Each Item must be noncontroversial:* This book exists to document widely agreed-upon standards, not to invent them. If a guideline is not appropriate in all cases, it will be presented that way (e.g., "Consider X…" instead of "Do X…") and we will note commonly accepted exceptions.
- *Each Item must be authoritative:* The guidelines in this book are backed up by references to existing published works. This book is intended to also provide an index into the C++ literature.
- *Each Item must need saying:* We chose not to define new guidelines for things that you'll do anyway, that are already enforced or detected by the compiler, or that are already covered under other Items.

  Example: "Don't return a pointer/reference to an automatic variable" is a good guideline, but we chose not to include it in this book because all of the compilers we tried already emit a warning for this, and so the issue is already covered under the broader Item 1, "Compile cleanly at high warning levels."

  Example: "Use an editor (or compiler, or debugger)" is a good guideline, but of course you'll use those tools anyway without being told; instead, we spend two of our first four Items on "Use an automated build system" and "Use a version control system."

  Example: "Don't abuse `goto`" is a great Item, but in our experience programmers universally know this, and it doesn't need saying any more.

Each Item is laid out as follows:

- *Item title:* The simplest meaningful sound bite we could come up with as a mnemonic for the rule.
- *Summary:* The most essential points, briefly stated.
- *Discussion:* An extended explanation of the guideline. This often includes brief rationale, but remember that the bulk of the rationale is intentionally left in the References.
- *Examples (if applicable):* Examples that demonstrate a rule or make it memorable.
- *Exceptions (if applicable):* Any (and usually rare) cases when a rule doesn't apply. But beware the trap of being too quick to think: "Oh, I'm special; this doesn't apply in my situation"—that rationalization is common, and commonly wrong.
- *References:* See these parts of the C++ literature for the full details and analysis.

In each section, we chose to nominate a "most valuable Item." Often, it's the first Item in a section, because we tried to put important Items up front in each part; but other times an important Item couldn't be put up front, for flow or readability reasons, and we felt the need to call it out for special attention in this way.

Acknowledgments

Many thanks to series editor Bjarne Stroustrup, to editors Peter Gordon and Debbie Lafferty, and to Tyrrell Albaugh, Kim Boedigheimer, John Fuller, Bernard Gaffney, Curt Johnson, Chanda Leary-Coutu, Charles Leddy, Heather Mullane, Chuti Prasertsith, Lara Wysong, and the rest of the Addison-Wesley team for their assistance and persistence during this project. They are a real pleasure to work with.

Inspiration for some of the "sound bites" came from many sources, including the playful style of [Cline99], the classic **import this** of [Peters99], and the legendary and eminently quotable Alan Perlis.

We especially want to thank the people whose technical feedback has helped to make many parts of this book better than they would otherwise have been. Series editor Bjarne Stroustrup's incisive comments from concept all the way through to the final draft were heavily influential and led to many improvements. We want to give special thanks to Dave Abrahams, Marshall Cline, Kevlin Henney, Howard Hinnant, Jim Hyslop, Nicolai Josuttis, Jon Kalb, Max Khesin, Stan Lippman, Scott Meyers, and Daveed Vandevoorde for their active participation in review cycles and detailed comments on several drafts of this material. Other valuable comments and feedback were contributed by Chuck Allison, Samir Bajaj, Marc Barbour, Travis Brown, Neal Coombes, Damian Dechev, Steve Dewhurst, Peter Dimov, Attila Feher, Alan Griffiths, Michi Henning, James Kanze, Mat Marcus, Petru Marginean, Robert C. "Uncle Bob" Martin, Bartosz Milewski, Balog Pal, Jeff Peil, Peter Pirkelbauer, Vladimir Prus, Dan Saks, Luke Wagner, Matthew Wilson, and Leor Zolman.

As usual, the remaining errors, omissions, and shameless puns are ours, not theirs.

Herb Sutter
Andrei Alexandrescu

Seattle, September 2004

This page intentionally left blank

1. Compile cleanly at high warning levels.

Summary

Take warnings to heart: Use your compiler's highest warning level. Require clean (warning-free) builds. Understand all warnings. Eliminate warnings by changing your code, not by reducing the warning level.

Discussion

Your compiler is your friend. If it issues a warning for a certain construct, often there's a potential problem in your code.

Successful builds should be silent (warning-free). If they aren't, you'll quickly get into the habit of skimming the output, and you *will* miss real problems. (See Item 2.)

To get rid of a warning: a) understand it; and then b) rephrase your code to eliminate the warning and make it clearer to both humans and compilers that the code does what you intended.

Do this even when the program seemed to run correctly in the first place. Do this even when you are positive that the warning is benign. Even benign warnings can obscure later warnings pointing to real dangers.

Examples

Example 1: A third-party header file. A library header file that you cannot change could contain a construct that causes (probably benign) warnings. Then wrap the file with your own version that **#include**s the original header and selectively turns off the noisy warnings for that scope only, and then **#include** your wrapper throughout the rest of your project. Example (note that the warning control syntax will vary from compiler to compiler):

```
// File: myproj/my_lambda.h -- wraps Boost's lambda.hpp
//   Always include this file; don't use lambda.hpp directly.
//   NOTE: Our build now automatically checks "grep lambda.hpp <srcfile>".
// Boost.Lambda produces noisy compiler warnings that we know are innocuous.
// When they fix it we'll remove the pragmas below, but this header will still exist.
//
#pragma warning(push)        // disable for this header only
  #pragma warning(disable:4512)
  #pragma warning(disable:4180)
  #include <boost/lambda/lambda.hpp>
#pragma warning(pop)         // restore original warning level
```

Example 2: "Unused function parameter." Check to make sure you really didn't mean to use the function parameter (e.g., it might be a placeholder for future expansion, or a required part of a standardized signature that your code has no use for). If it's not needed, simply delete the name of a function parameter:

```
// ... inside a user-defined allocator that has no use for the hint ...

// warning: "unused parameter 'localityHint'"
pointer allocate( size_type numObjects, const void *localityHint = 0 ) {
  return static_cast<pointer>( mallocShared( numObjects * sizeof(T) ) );
}

// new version: eliminates warning
pointer allocate( size_type numObjects, const void * /* localityHint */ = 0 ) {
  return static_cast<pointer>( mallocShared( numObjects * sizeof(T) )   );
}
```

Example 3: "Variable defined but never used." Check to make sure you really didn't mean to reference the variable. (An RAII stack-based object often causes this warning spuriously; see Item 13.) If it's not needed, often you can silence the compiler by inserting an evaluation of the variable itself as an expression (this evaluation won't impact run-time speed):

```
// warning: "variable 'lock' is defined but never used"
void Fun() {
  Lock lock;

  // ...

}

// new version: probably eliminates warning
void Fun() {
  Lock lock;
  lock;

  // ...

}
```

Example 4: "Variable may be used without being initialized." Initialize the variable (see Item 19).

Example 5: "Missing **return***."* Sometimes the compiler asks for a **return** statement even though your control flow can never reach the end of the function (e.g., infinite loop, **throw** statements, other **return**s). This can be a good thing, because sometimes you only *think* that control can't run off the end. For example, **switch** statements that do not have a **default** are not resilient to change and should have a **default** case that does **assert(false)** (see also Items 68 and 90):

```
// warning: missing "return"
int Fun( Color c ) {
  switch( c ) {
  case Red:     return 2;
  case Green:   return 0;
  case Blue:
  case Black:   return 1;
  }
}

// new version: eliminates warning
int Fun( Color c ) {
  switch( c ) {
  case Red:     return 2;
  case Green:   return 0;
  case Blue:
  case Black:   return 1;
  default:      assert( !"should never get here!" );    // !"string" evaluates to false
                return -1;
  }
}
```

Example 6: "Signed/unsigned mismatch." It is usually not necessary to compare or assign integers with different signedness. Change the types of the variables being compared so that the types agree. In the worst case, insert an explicit cast. (The compiler inserts that cast for you anyway, and warns you about doing it, so you're better off putting it out in the open.)

Exceptions

Sometimes, a compiler may emit a tedious or even spurious warning (i.e., one that is mere noise) but offer no way to turn it off, and it might be infeasible or unproductive busywork to rephrase the code to silence the warning. In these rare cases, as a team decision, avoid tediously working around a warning that is merely tedious: Disable that specific warning only, disable it as locally as possible, and write a clear comment documenting why it was necessary.

References

[Meyers97] §48 • [Stroustrup94] §2.6.2

25. Take parameters appropriately by value, (smart) pointer, or reference.

Summary

Parameterize well: Distinguish among input, output, and input/output parameters, and between value and reference parameters. Take them appropriately.

Discussion

Choosing well among values, references, and pointers for parameters is good habit that maximizes both safety and efficiency.

Although efficiency should not be our primary up-front concern (see Item 8), neither should we write needlessly inefficient code when all other things, including clarity, are equal (see Item 9).

Prefer to follow these guidelines for choosing how to take parameters. For input-only parameters:

- Always **const**-qualify all pointers or references to input-only parameters.
- Prefer taking inputs of primitive types (e.g., **char**, **float**) and value objects that are cheap to copy (e.g., **Point**, **complex<float>**) by value.
- Prefer taking inputs of other user-defined types by reference to **const**.
- Consider pass-by-value instead of reference if the function requires a copy of its argument. This is conceptually identical to taking a reference to **const** plus doing a copy, and it can help compiler to better optimize away temporaries.

For output or input/output parameters:

- Prefer passing by (smart) pointer if the argument is optional (so callers can pass null as a “not available” or “don’t care” value) or if the function stores a copy of the pointer or otherwise manipulates ownership of the argument.
- Prefer passing by reference if the argument is required and the function won’t store a pointer to it or otherwise affect its ownership. This states that the argument is required and makes the caller responsible for providing a valid object.

Don’t use C-style varargs (see Item 98).

References

[Alexandrescu03a] • [Cline99] §2.10-11, 14.02-12, 32.08 • [Dewhurst03] §57 • [Koenig97] §4 • [Lakos96] §9.1.11-12 • [McConnell93] §5.7 • [Meyers97] §21-22 • [Stroustrup94] §11.4.4 • [Stroustrup00] §5.5, §11.6, §16.3.4 • [Sutter00] §6, §46

73. Throw by value, catch by reference.

Summary

Learn to **catch** properly: Throw exceptions by value (not pointer) and catch them by reference (usually to **const**). This is the combination that meshes best with exception semantics. When rethrowing the same exception, prefer just **throw;** to **throw e;**.

Discussion

When throwing an exception, throw an object by value. Avoid throwing a pointer, because if you throw a pointer, you need to deal with memory management issues: You can't throw a pointer to a stack-allocated value because the stack will be unwound before the pointer reaches the call site. You could throw a pointer to dynamically allocated memory (if the error you're reporting isn't "out of memory" to begin with), but you've put the burden on the catch site to deallocate the memory. If you feel you really must throw a pointer, consider throwing a value-like smart pointer such as a **shared_ptr<T>** instead of a plain **T***.

Throwing by value enjoys the best of all worlds because the compiler itself takes responsibility for the intricate process of managing memory for the exception object. All you need to take care of is ensuring that you implement a non-throwing copy constructor for your exception classes (see Item 32).

Unless you are throwing a smart pointer, which already adds an indirection that preserves polymorphism, catching by reference is the only good way to go. Catching a plain value by value results in slicing at the catch site (see Item 54), which violently strips away the normally-vital polymorphic qualities of the exception object. Catching by reference preserves the polymorphism of the exception object.

When rethrowing an exception **e**, prefer writing just **throw;** instead of **throw e;** because the first form always preserves polymorphism of the rethrown object.

Examples

Example: Rethrowing a modified exception. Prefer to rethrow using **throw;**:

```
catch( MyException& e ) {                      // catch by reference to non-const
  e.AppendContext("Passed through here");     // modify
  throw;                                       // rethrow modified object
}
```

References

[Dewhurst03] §64-65 • [Meyers96] §13 • [Stroustrup00] §14.3 • [Vandevoorde03] §20

74. Report, handle, and translate errors appropriately.

Summary

Know when to say when: Report errors at the point they are detected and identified as errors. Handle or translate each error at the nearest level that can do it correctly.

Discussion

Report an error (e.g., write **throw**) wherever a function detects an error that it cannot resolve itself and that makes it impossible for the function to continue execution. (See Item 70.)

Handle an error (e.g., write a **catch** that doesn't rethrow the same or another exception or emit another kind of error code) in the places that have sufficient knowledge to handle the error, including to enforce boundaries defined in the error policy (e.g., on **main** and thread mainlines; see Item 62) and to absorb errors in the bodies of destructors and deallocation operations.

Translate an error (e.g., write a **catch** that does rethrow a different exception or emits another kind of error code) in these circumstances:

- *To add higher-level semantic meaning:* For example, in a word processing application, **Document::Open** could accept a low-level unexpected-end-of-file error and translate it to a document-invalid-or-corrupt error, adding semantic information.
- *To change the error handling mechanism:* For example, in a module that uses exceptions internally but whose C API public boundary reports error codes, a boundary API would catch an exception and emit a corresponding error code that fulfills the module's contract and that the caller can understand.

Code should not accept an error if it doesn't have the context to do something useful about that error. If a function isn't going to handle (or translate, or deliberately absorb) an error itself, it should allow or enable the error to propagate up to a caller who can handle it.

Exceptions

It can occasionally be useful to accept and re-emit (e.g., **catch** and rethrow) the same error in order to add instrumentation, even though the error is not actually being handled.

References

[Stroustrup00] §3.7.2, §14.7, §14.9 • [Sutter00] §8 • [Sutter04] §11 • [Sutter04b]

83. Use a checked STL implementation.

Summary

Safety first (see Item 6): Use a checked STL implementation, even if it's only available for one of your compiler platforms, and even if it's only used during pre-release testing.

Discussion

Just like pointer mistakes, iterator mistakes are far too easy to make and will usually silently compile but then crash (at best) or appear to work (at worst). Even though your compiler doesn't catch the mistakes, you don't have to rely on "correction by visual inspection," and shouldn't: Tools exist. Use them.

Some STL mistakes are distressingly common even for experienced programmers:

- *Using an invalidated or uninitialized iterator:* The former in particular is easy to do.
- *Passing an out-of-bounds index:* For example, accessing element 113 of a 100-element container.
- *Using an iterator "range" that isn't really a range:* Passing two iterators where the first doesn't precede the second, or that don't both refer into the same container.
- *Passing an invalid iterator position:* Calling a container member function that takes an iterator position, such as the position passed to **insert**, but passing an iterator that refers into a different container.
- *Using an invalid ordering:* Providing an invalid ordering rule for ordering an associative container or as a comparison criterion with the sorting algorithms. (See [Meyers01] §21 for examples.) Without a checked STL, these would typically manifest at run time as erratic behavior or infinite loops, not as hard errors.

Most checked STL implementations detect these errors automatically, by adding extra debugging and housekeeping information to containers and iterators. For example, an iterator can remember the container it refers into, and a container can remember all outstanding iterators into itself so that it can mark the appropriate iterators as invalid as they become invalidated. Of course, this makes for fatter iterators, containers with extra state, and some extra work every time you modify the container. But it's worth it—at least during testing, and perhaps even during release (remember Item 8; don't disable valuable checks for performance reasons unless and until you know performance is an issue in the affected cases).

Even if you don't ship with checking turned on, and even if you only have a checked STL on one of your target platforms, at minimum ensure that you routinely run your full complement of tests against a version of your application built with a checked STL. You'll be glad you did.

Examples

Example 1: Using an invalid iterator. It's easy to forget when iterators are invalidated and use an invalid iterator (see Item 99). Consider this example adapted from [Meyers01] that inserts elements at the front of a **deque**:

```
deque<double>::iterator current = d.begin();

for( size_t i = 0; i < max; ++i )
  d.insert( current++, data[i] + 41 );          // do you see the bug?
```

Quick: Do you see the bug? You have three seconds.—Ding! If you didn't get it in time, don't worry; it's a subtle and understandable mistake. A checked STL implementation will detect this error for you on the second loop iteration so that you don't need to rely on your unaided visual acuity. (For a fixed version of this code, and superior alternatives to such a naked loop, see Item 84.)

Example 2: Using an iterator range that isn't really a range. An iterator range is a pair of iterators **first** and **last** that refer to the first element and the one-past-the-end-th element of the range, respectively. It is required that **last** be reachable from **first** by repeated increments of **first**. There are two common ways to accidentally try to use an iterator range that isn't actually a range: The first way arises when the two iterators that delimit the range point into the same container, but the first iterator doesn't actually precede the second:

```
for_each( c.end(), c.begin(), Something );     // not always this obvious
```

On each iteration of its internal loop, **for_each** will compare the first iterator with the second for equality, and as long as they are not equal it will continue to increment the first iterator. Of course, no matter how many times you increment the first iterator, it will never equal the second, so the loop is essentially endless. In practice, this will, at best, fall off the end of the container **c** and crash immediately with a memory protection fault. At worst, it will just fall off the end into uncharted memory and possibly read or change values that aren't part of the container. It's not that much different in principle from our infamous and eminently attackable friend the buffer overrun.

The second common case arises when the iterators point into different containers:

```
for_each( c.begin(), d.end(), Something );     // not always this obvious
```

The results are similar. Because checked STL iterators remember the containers that they refer into, they can detect such run-time errors.

References

[Dinkumware-Safe] • [Horstmann95] • [Josuttis99] §5.11.1 • [Metrowerks] • [Meyers01] §21, §50 • [STLport-Debug] • [Stroustrup00] §18.3.1, §19.3.1

This page intentionally left blank

Index

#include
  and using, 108
  vs. forward declaration, 40
#include guards, 27, 33
  internal vs. external, 43
#undef
  as soon as possible, 33
&&
  preferable to nested ifs, 38
?:, 36
[]. *See* operators, []
++C, 50

A

Abelson, Harold, 13
Abrahams, Dave, xv
abstraction, 20
  and dependency management, 11
  and get/set, 20, 72, 73
  and interfaces, 62
abstractions
  build higher-level from lower-level, 12
  depending upon instead of details, 41
  vs. details, 128
accumulate, 125
Acyclic Visitor, 41
ADL, 104, 105, 106, 107, 122
  and template customization, 122
  disabling unwanted, 124
aggregates, 20
Albaugh, Tyrrell, xv
algorithmic complexity, 14
  and STL, 14
  exponential, 15
  linear-looking that is really quadratic, 15, 156
algorithms
  and design patterns, 162
  are loops, 159
  binary_search, 165
  count, 165
  count_if, 165
  equal_range, 165
  find, 165
  find_if, 165
  lower_bound, 165
  nth_element, 166
  partial_sort, 166
  partial_sort_copy, 166
  partition, 166
  searching, 165
  sort, 166
  sorting, 166
  stable_partition, 166
  stable_sort, 166
  upper_bound, 165
  vs. loops, 38, 162
alignment, 176
Allison, Chuck, xv
allocation, 111
  never allocate more than once per statement, 25
allocator
  example use of, 5
ambiguities, 77
ambiguities,
  avoiding declaration, 13
amortized constant time, 155
append, 135
arithmetic operators. *See* operators, arithmetic
arrays
  fixed-size, 15
  inferior to containers, 152
assert, 33, 130, 135
  example of, 5, 98, 175
  macro needed for, 33
  only for internal programming errors, 132, 134
  prefer instead of logic_error, 131
assertions. *See* assert
assignment
  copy. *See* copy assignment
  self, 99, 138
assignment operators. *See* operators, assignment
asymptotic complexity. *See* algorithmic complexity
at
  vs. [], 136
atomic operations, 21
auto_ptr, 94, 154

B

Bajaj, Samir, xv
BankAccount, 72
Barbour, Marc, xv
base classes. *See* classes, base
base two, 176
basic_string, 12, *See also* containers
  append, 135
  find_first_of, 136
  insert, 135
  monolithic, 79
behavior
  undefined. *See* undefined behavior
Bell, Gordon, 13
Bentley, Jon, 13, 16
BetweenValues, 164
Big Four, 55, 85, 94, *See also* default constructor; copy construction; copy assignment; destructor
Big-Oh. *See* algorithmic complexity
binary compatibility, 116, 120
binary_function, 172
binary_search, 165
bind2nd, 162, 163
  example use of, 163, 164
Bird, 67
bloat, 112
Boedigheimer, Kim, xv
Boost, 3, 147, *See also* shared_ptr
  discriminated unions library, 121
  format library, 184
  Lambda library, 4, 162, 163, 164
  Lambda library, example use of, 163
  preprocessor library, 33
bounds checking, 29, 152
brace placement, 2
braces. *See* brace placement
  matching, 38
branch prediction, 16
Bridge, 162
Brown, Travis, xv
buffer overruns. *See* security
bugs. *See* insects
build
  breaking, 8
  unit tests, 8
build system
  automated, 7
build times, 76

C

C, 36, *See also* C, obsolete uses of
C, obsolete uses of, xi
  arrays, 37, 152, 186
  casts, 180, 181
  global namespace, 108
  Hungarian notation, 3
  implicit cast from const char[] to (non-const) char* hole in the type system, 179
  macros, 32, 33
  manual memory management, 24, 152
  manual resource management, 24, 152
  memcpy/memcmp (except for PODs), 182
  null-terminated character array strings, 37, 152
  pointer arithmetic, 152
  printf, 184
  realloc, 12
  sprintf, 184
  switching on a type flag, 174, 175
  unions to reinterpret representation, 183
  unsafe functions (strcpy/strncpy, strcmp, sprintf, gets, etc.), 185
  varargs, 46, 184
  variable definition at beginning of scope, 35, 36
C++
  vs. ++C, 50
caching, 16
caffeine
  lack of, 96
callback functions, 133
  and exceptions, 114
  instead of locking, 23
Carlson, Richard
  reference to, 2, 144, 155
casts, 180
  and not const, 179
  explicit preferred, 6
catch
  ..., 81, 93, 114, 115, 133, 140
Catch-22, 127
cerr, 19, 113
char_traits, 125
check in. *See* version control system
check out. *See* version control system
checked STL implementation, 160
checked_cast, 178
cin, 19, 113
clarity
  prime importance of, 13
class templates. *See also* templates
  specialization, 127
classes
  and namespaces, 104
  and nonmember functions, 104
  and portability, 116
  base, 56, 69, 90, 91, 96, 101
  composition vs. inheritance, 58, 61
  concrete, 60, 91
  data members, 72
  derived. *See* polymorphism *and* substitutability
  exception, 56
  kinds of, 56
  minimal vs. monolithic, 57
  mixin, 65
  policy, 56, 65, 91
  traits, 56

unions, 183
value, 56, 101, 154
clean compiles. *See* compiler warnings
clear
better than cute, 13
cliff, 85
Cline, Marshall, xv
clog, 113
Clone, 96, 97
vs. copy construction, 97
Cobol, 36
code reviews, 9
this book's table of contents as checklist, 9
coding style
vs. design style, 11
cohesion, 12, 38
COM, 7, 63, 91, 115, 133
Command, 41, 121
comments, 2
CompareThings, 171
compatibility
source vs. binary, 73
compile
cleanly. *See* compiler warnings
compile time
and errors, 28
compiler firewall. *See* Pimpl
compiler warnings, 4
compiler-generated functions, 85, *See* copy construction; copy assignment; destructor
compile-time
conditions, 29
errors, 27
polymorphism, 29
complex
simple better than, 13
complexity
algorithmic. *See* algorithmic complexity
asymptotic. *See* algorithmic complexity
compose, 163
compose2
example use of, 164
composition
vs. inheritance, 58, 61
concurrency, 19, 21, *See also* locking
vast majority of objects not shared, 22
conditional compilation, 33
conditions
compile-time, 29
const, 27, 30
and pointers, 30
avoid on pass-by-value parameters, 31
instead of magic numbers, 34
not deep, 30
simplifies code, 30
viral, 30
const_cast, 179
const-correctness, 31, 128, 179
construction
copy. *See* copy construction
construction order
of member variables, 86
ConstructionWasOK
not recommended, 141
constructor parameters
prefer named variables instead of temporaries, 13
constructors
and virtual functions, 88
copy. *See* copy construction
default. *See* default constructor
initialization list, 87
initialization list ordering not significant, definition order significant, 86
post-constructors, 88
prefer initializer list instead of assignment, 18
reporting errors from, 141, 142
virtual constructors, 88
containers
and copy construction/assignment, 95
and smart pointers, 95
and thread safety, 21
choosing, 150
default, 150
hash-based, 15, 150, 181
heterogeneous, 154
index, 154
map, and optional values, 154
of non-value types, 154
range vs. single-element functions, 155, 156
shrink-to-fit, 157
store values, 154
string, 152
vector, 150, 152, 153
vector vs. list, 151
vector, advantages of, 150
conversion sequences, 70
conversions
implicit, 70, *See* implicit type conversions
named functions, 70
Coombes, Neal, xv
copy, 107
copy assignment, 25, 55, 85, 87, 99
and containers, 95
and copy construction, 94, 95
and destructor, 94
and swap, 101
not virtual, 99
copy construction, 25, 55, 85
copy constructors
and containers, 95
and copy assignment, 94, 95
and destructor, 94
vs. Clone, 97
copy-on-write, 23
CORBA, 7, 63, 91, 115, 133
correct
better than fast, 13
correctness
prime importance of, 13
corruption, 21
count, 165
count_if, 165
coupling, 19

cout, 19, 113
covariance, 69
COW. *See* copy-on-write
CPU-bound, 17
Create, 89
curly braces. *See* brace placement
CustomAllocator, 80
customization
  and C++ standard library, 125
  of templates, 122
CustomString, 117
cute
  clear better than, 13
cvs, 8
cyclic dependencies, 40
  breaking, 41

D

dangling pointers, 185
data
  exposing, 20
  global. *See* global variables
data validation, 29
data volumes
  growth of, 14
database-bound, 17
Date, 72
deadlock, 21
deallocation, 111
deallocation functions
  never fail, 92
Dechev, Damian, xv
declaration
  vs. definition, 40
declaration ambiguities
  avoiding, 13
default, 175
default arguments
  and virtual functions, 66
default constructor, 55, 85, 87, 156
default container
  vector, 150
definition
  of member variables, 86
  vs. declaration, 40
delete. *See also* operators, delete
  and polymorphism, 91
  with new, 80
dependencies, 103
  and templates, 42
  compile-time, 58
  cyclic. *See* cyclic dependencies
  managing, 20
  upon abstractions instead of details, 41
dependency cycles
  across modules, 41
Dependency Inversion Principle, 41, 62
dependency management, 74, *See also* encapsulation *and* information hiding
  broad importance of, 11
  member vs. nonmember functions, 79
dependent names, 125
deployment
  ease of, 57
design patters
  and algorithms, 162
design style
  design vs. coding style, 11
destructor, 55, 85
  and copy assignment, 94
  and copy construction, 94
  nonvirtual, 61, *See also* slicing
  public and virtual, 63
destructors, 68, *See also* RAII
  and exceptions, 115
  and virtual functions, 88
  in base classes, 90
  never fail, 92
details
  vs. abstractions, 128
Dewhurst, Steve, xv
Diamond, Norman, 85
Dimov, Peter, xv
dint
  gratuitous use of odd word, 162
disabling warnings. *See* warnings
disk capacity
  growth of, 14
disk-bound, 17
distance, 107, 156, 165
divide and conquer. *See* minimal vs. monolithic
DLLs, 103
DoClone, 98
downcasts, 29
Draw, 175
dusty corners, 13
dynamic_cast, 69, 178
  downcasting with, 29
dynamically checked errors. *See* errors, dynamic checking

E

EBO. *See* empty base class optimization
ECO. *See* empty base class optimization
efficiency. *See* performance
empty base class optimization, 59, 63
empty()
  vs. size() == 0, 128
encapsulation, 20, 57, 72, 74, 76
  member vs. nonmember functions, 79
enums, 29, 175
  instead of magic numbers, 34
equal_range, 165
ER units
  comparison with, xiii
errno, 140, *See also* error codes
error code
  overuse, 142
error codes
  translating to/from exceptions, 115
  vs. exceptions, 140
error handling policy. *See* errors, policy for handling
error messages

and macros, 33
error safety, 57, 59, 77
and RAII, 24
errors
and modules, 133
and operators, 141
assert, 130
categorizing, 133
compile-time, 28
constructors, 141
detection, 133
dynamic checking, 28
handling, 133, 145
identifying, 132
ignoring, dangers of, 140
internal assumptions, 130
invariants to test for. *See* invariants
link-time, 28
policy for handling, 132
prefer compile- and link-time to run-time, 27, 28
propagating, 140
propagation, 133
reporting, 133, 145
retrying, 138
run-time, 132
severity, 133
static checking, 28
translating, 144, 145
vs. non-errors, 134
error-safety, 150
basic guarantee, 137
copy construction, 99
no-fail guarantee, 137
not penalizing code that doesn't need stronger guarantees, 137
strong guarantee, 137
evil
root of all, 11
exception
what, 147
exception classes. *See* classes, exception
exception handling. *See also* errors; error-safety
catch by reference, 144
overuse, 142
throw by value, 144
warning against disabling, 143
exception safety. *See* error safety
exception specifications, 93, 146
avoid, 146
static vs. dynamic checking, 147
exceptions
and callback functions, 114
and destructors, 115
and main, 114
and modules, 114
and slicing, 144
and threads, 114
not across module boundaries, 114
translating to/from error codes, 115
vs. error codes, 140
explicit, 70, 97
explicit loops
fewer in STL-using programs, 162
explicit qualification, 77, 110
expression templates, 50, 53
external locking, 22

F

facets
mistakes of, 121
factory
example use of, 89
Factory, 162
factory functions, 19
fast
correct better than, 13
Feher, Attila, xv
File, 72, 136
find, 18, 165
find_first_of, 136, 142
find_if, 165, 169
FlagNth, 169
Fly, 67
fools, 11
for_each, 15, 162
example use of, 161
formatting, 2
Fortran, 36
forward declaration
vs. #include, 40
French
grauitous use of, 51
friend, 55
fudgeFactor, 112
full build, 7, *See also* build system
Fuller, John, xv
function
to avoid uninitialized variables, 37
unit of work, 134
function arguments
order of evaluation, 54
function objects, 162, *See also* predicates
example use of, 164
vs. functions, 170
writing correctly, 172
function parameters, 45
and binders, 162
and compile-time dependencies, 76
and const, 31, 46
and conversions, 48
and copying, 46
and null, 46
and preconditions, 134
and primitive types, 46
and smart pointers, 46
and unary_function/binary_function, 170
and user-defined types, 46
and varargs, 46
in constructors, 89
input, 46
output, 46
pass by value vs. pass by reference, 46
unary and binary operators, 48
function templates, 113

and not specialization, 126
and overload resolution, 126
functions
compiler-generated, 85
deallocation, 92
length, 38
member vs. nonmember, 48, 79
nesting, 38
vs. function objects, 170
functions,compiler-generated. *See* default constructor; copy construction; copy assignment; destructor

G

Gaffney, Bernard, xv
generic programming. *See* templates
geniuses, 11
get/set, 73
and abstraction, 20, 72, 73
GetBuffer, 75
GetBuilding, 66
GetLastError, 140
getstr, 53
global data. *See* global variables
global state. *See* global variables
global variables, 19, 39
and dependency management, 11
initialization of, 19
limit parallelism, 19
Gordon, Peter, xv
greater
example use of, 164
grep, 181
Griffiths, Alan, xv
guarantees
for error safety. *See* error-safety

H

handles
to internal data, 74
hash-based containers. *See* containers, hash-based
Haskell, 28
header files
self-sufficient, 42
wrapping third-party headers, 4
header guards. *See* #include guards
headers
and linkage, 112
and not unnamed namespaces, 113
and static, 113
precompiled, 42
Henney, Kevlin, xv
Henning, Michi, xv
heterogeneous containers, 154
hide information. *See* information hiding
hiding
names, 66, 82
hijacking
and macros, 32
Hinnant, Howard, xv
Hoare, C.A., 16
Hungarian notation, 3
hygiene
and not macros, 32
Hyslop, Jim, xv

I

implicit conversions, 70
benefits of, 71
dangers of, 71
implicit interface, 122
and customization, 122
implicit type conversions
avoided by overloading, 51
import this, xv
incremental build, 7, *See also* build system
indentation, 2
index containers, 154
indexing
vs. iterators, 128
information hiding, 72
and dependency management, 11
inheritance
and dependency management, 11
and reuse, 64
misuse of, 64
not from concrete base classes, 60
public, 64
vs. composition, 58, 61
initialization
and constructors, 87
default, 87
of global variables, 19
of member variables, 86
of variables, 35, 36
static vs. dynamic, 39
variables. *See* variable, not initialized
zero, 39
initialization dependencies, 39
inline, 17, 113
and profiler, 17
in- XE "new" \t "*See also* operators, new" XE "delete" \t "*See also* operators, delete" place new. *See* new
insects, 9, 12, 28, 30, 35, 36, 39, 52, 81, 137
insert, 135, 139, 156
at a specific location, 150
inserter
example use of, 163
interface
implicit. *See* implicit interface
Interface Principle, 104
interfaces
abstract, 62
intermittent crashes, 36
internal locking, 22
internals
exposing, 20
invalid iterators, 185
invariants, 18, 20, 28, 64, 72, 73, 74, 130, 131, 132, 134, 135, 136, 137, 138, 140, 141, 142

iostreams, 113
is_in_klingon, 61
is-a. *See* substitutability, *See* substitutability
IsHeavy, 170
iterator ranges, 161
iterator_traits, 125
iterators, 151
  comparing with != instead of <, 128
  invalid, 161, 185
  ranges, 161
  vs. indexing, 128

J

Java, 28, 147
Johnson, Curt, xv
Josuttis, Nicolai, xv
juggling, 152

K

K&R style. *See* brace placement
Kalb, Jon, xv
Kanze, James, xv
Kernighan, Brian, 173
Khesin, Max, xv
KISS, 13
Knuth, Donald, 11, 16
Koenig lookup. *See* ADL

L

Lafferty, Debbie, xv
Lambda library. See Boost, Lambda library
land mines, 27
Last Word
  not this book, xii
Latin
  gratuitous use of, 59, 141
LaunchSatellite, 139
Law of Second Chances, 63
leak
  memory, 81
leaks, 137
Leary-Coutu, Chanda, xv
Leddy, Charles, xv
length
  of lines, 2
less
  example use of, 164
libraries
  shared, 103
lifetime. *See* object lifetime
line length, 2
link time
  and errors, 27, 28
linkage
  and headers, 112
  external, 19
Lippman, Stan, xv
Liskov Substitution Principle. *See* substitutability
Lisp, 28
list. *See also* containers
  vs. vector, 151
literals
  and magic numbers. *See* magic numbers
livelock, 21
locality of reference, 151
localized_string, 61
locking
  external, 22
  in increasing address order, 23
  internal, 22
  lock-free designs, 23
  not needed for immutable objects, 23
  using callback functions instead of, 23
logic_error
  example of, 5
  prefer assertions instead of, 131
lookup
  two-phase, 125
loops
  fewer explicit loops in STL-using programs, 162
  vs. algorithms, 162
lower_bound, 165

M

macros, 27, 32
  and conditional compilation, 33
  interfering with template instantiations, 33
  to enable/disable threading support, 23
magic numbers, 34
main
  and exceptions, 114
make, 7, *See also* build system
malloc, 131
managing dependencies, 103, *See* dependency management
Marcus, Matt, xv
Marginean, Petru, xv
Martin, Robert C., xv
Matrix, 57, 72
MAX_PATH, 37
McConnell, Steve, 13, 130
mem_fun, 170
mem_fun_ref, 170
member variables
  public vs. private, 72
member vs. nonmember functions, 79
memcmp, 182
memcpy, 182
memory leaks, 81
memory management
  and containers, 152
memory-bound, 17
MemoryPool, 82
Meyers, Scott, xv
Milewski, Bartosz, xv
Ming vases, 152
minimal vs. monolithic, 55, 57
missing return. *See* return, missing
mixin classes. *See* classes, mixin
ML, 28
modules
  allocating and deallocating memory in same, 111
  and error handling, 133

and exceptions, 114
and not exceptions, 114
defined, 103
interdependence between, 40
interfaces use only sufficiently portable types, 116
monolithic classes, 79
monolithic vs. minimal, 55, 57
Moore's Law, 14
Mullane, Heather, xv
mutable, 30

N

name hiding, 66, 82
name lookup, 77
two-phase, 125
named variables
prefer as constructor parameters, 13
names
dependent, 125
symbolic vs. magic numbers, 34
namespaces, 103
and using, 108
pollution of, 19, 108, 109, 110
type and its nonmember functions in same, 104
type and unrelated functions in separate, 106
unnamed. *See* unnamed namespace
using, 108
naming
and macros, 33
variables. *See* Hungarian notation
naming convention, 2
NDEBUG, 111, 130
Nefarious, 92, 93
nesting, 38
network-bound, 17
new, 141, *See also* operators, new
immediately giving result to an owning object, 25
in-place, 82
never allocate more than once per statement, 25
nothrow, 82
with delete, 80
nifty counters, 113
Node, 73
nongeneric code
unintentionally, 128
Nonvirtual Interface pattern, 68, 69, 90, 98
not1, 170
nothrow new. *See* new
nth_element, 166
example use of, 167
NVI. *See* Nonvirtual Interface pattern

O

object lifetime
minimizing, 35
objects
temporary. *See* temporary objects
Observer, 162
obsolete practices, 2, *See* C, obsolete uses of
external #include guards, 43
Hungarian notation, 3
SESE. *See* single entry single exit
Occam, William of, 51
ODR. *See* one definition rule
offsetof, 176
ointment
fly in the, 81
one definition rule, 110
operator delete
never fails, 92
operator overloading
gratuitous, 13
preserve natural semantics, 47
operators, 45
&&, 52
(), 168
,, 52
[], 135, 136
[] vs. iterators, 128
| |, 52
++, 17, 18, 50
and ADL, 105
and namespaces, 104, 105
arithmetic, 48
assignment, 48, 78, 93
binary, 48
const char* (on strings), 71
copy assignment. *See* copy assignment
decrement, 50
delete, 80, 82, 93, 111
increment, 50
member vs. nonmember, 48
new, 80, 82, 111, 141
overloaded, 47
preserve natural semantics, 47, 48, 50
reporting errors from, 141
optimization. *See also*
temporary objects, *See also*
temporary objects, *See also*
temporary objects, *See also*
temporary objects
and exception specifications, 146
and inline, 17
and libraries, 17
by using STL, 18
compile-time evaluation, 121
copy-on-write outdated, 157
empty base class, 63
enabling compiler's, 49, 99
encapsulate where possible, 17
in STL implementations, 94
indexing vs. iteration, 128
must be based on measurement, 16
prefer improving algorithmic complexity over micro-optimizations, 17
premature, 13, 14, 15, 16, 17, 18, 50, 51, 59, 87, 171

range vs. single-element functions, 156
self-assignment check, 138
static binding, 121
optional values
and map, 154
order dependencies, 19, 23, 25, 39, 52, 53, 54, 69, 86, 109, 110, 124, 169, 176
Ostrich, 67
out_of_range, 136
overload resolution, 77
overloading
and conversions, 70
and function templates, 126
of operators, 13
to avoid implicit type conversions, 51
overriding, 66

P

pair, 56
Pal, Balog, xv
parameters
pass by value vs. pass by reference, 18
unused. *See* unused parameters
partial specialization. *See* specialization, partial
partial_sort, 166
example use of, 167
partial_sort_copy, 166
partition, 162, 166
example use of, 166
Pascal, 36
Peil, Jeff, xv
pejorative language
and macros, 32
performance, 28, 141
Perlis, Alan, xi, xv, 11, 27, 45, 60, 103, 129, 173
personal taste
matters of, 2
pessimization, 18
Pimpl, 30, 58, 69, 72, 76, 78, 101, 172, *See also* encapsulation
and dependency management
and shared_ptr, 78
pipelining, 16
Pirkelbauer, Peter, xv
placement
of braces. *See* brace placement
plain old data. *See* POD
platform-dependent operations
wrapping, 21
Plauger, P.J., 173
plus, 162, 163
example use of, 163
POD, 176, 183
pointer_to_unary_function, 170
pointers
and const, 30
and not static_cast, 178
dangling, 185
points of customization. *See* customization
policy classes. *See* classes, policy
policy-based design, 63
pollution (of names and namespaces), 19, 35, 108, 109, 110
polymorphism, 66
ad-hoc, 120
and delete, 91
and destruction, 90
and not arrays, 186
and slicing, 96
compile-time vs. run-time, 29
controlled, 59
dynamic, 128
dynamic, 64, 120
inclusion, 120
static, 63, 120
static and dynamic, 120, 175
static vs. dynamic, 65
vs. slicing, 144
vs. switch on type tag, 38
vs. switching on type, 174
Port, 24
portable types
and module interfaces, 116
postconditions, 66, 69, 124, 130, 131, 134, 135, 136, 138, 140, 142
and virtual functions, 66
post-constructors, 88
PostInitialize, 89
pragmatists, 11
Prasertsith, Chuti, xv
precompiled headers, 42
preconditions, 66, 69, 132, 134, 135, 136, 142
and virtual functions, 66
predicates. *See also* function objects
pure functions, 168
premature optimization. *See* optimization, premature
pressure
schedule pressure, xiii
priority_queue, 166
processes
multiple, 21
profiler
and inline, 17
using. *See* optimization
proverbs
Chinese, 8
German, 177
Latin, 16, 156
level of indirection, 126
Romanian, 177
Prus, Vladimir, xv
ptr_fun, 170
public data, 20
push_back, 15, 155
Python, 28

Q

qualification
explicit, 77
qualification, explicit, 110
qualified
vs. unqualified, 123

R

race conditions, 21
RAII, 5, 24, 38, 56, 94, 95
  and copy assignment, 25
  and copy construction, 25
range checking, 135
ranges
  of iterators, 161
realloc, 12
Rectangle, 64
recursive search
  not reporting result using exception, 142
reference counting, 157
registry
  factory and, 19
reinterpret_cast, 177, 180, 181, 183, 184, 185
release
  unit of. *See* module
reliability, 27
remove_copy_if, 169
remove_if, 169
replace_if, 162
resource acquisition is initialization. *See* RAII
resource management, 94, *See also* RAII
  and constructors, 87
  and RAII, 24
  and smart pointers, 24
  never allocate more than once per statement, 25
  resources should be owned by objects, 25
resources. *See* resource management
responsibility
  growth, 12
  of an entity, 12
restricted values
  of integers, 29
return
  missing, 5
reuse
  and inheritance, 64
reviews
  of code. *See* code reviews
ripple effect, 20
root of all evil, 11
Ruby, 28
run time
  and errors, 27, 28

S

safety, 27
Saks, Dan, xv
scalability
  coding for, 14
schedule pressure, xiii
Schwarz counters, 113
Schwarz, Jerry, 113
Second Chances
  Law of, 63
security, 15
  and checked STL implementation, 160
  and exception handling performance, 142
  arrays and, 15
  buffers, 152
  pointers, 152
  printf, 184
  ssh, 8
  strcpy, 185
Security, 72
self-assignment, 99, 138
self-sufficient header files, 42
serialization
  of access to shared objects, 21
SESE. *See* single entry single exit
shallow const, 30
Shape, 175
shared libraries, 103
shared state
  and dependency management, 11
shared_ptr, 111, 121, 149
  and arrays, 186
  and containers, 154
  and modules, 111
  and optional values in maps, 154
  and overuse, 25
  and Pimpl, 78
  example use of, 24, 25, 76, 78, 89, 182
  throwing, 144
shared_ptr, 149
shared_ptr, 172
sheep's clothing, 39
shrink-to-fit, 157
signed/unsigned mismatch, 6
simple
  better than complex, 13
simplicity
  prime importance of, 13
single entry single exit, 3
Singleton, 39
skins, 139
slicing, 61, 96
  and polymorphism, 96
  of exceptions, 144
Smalltalk, 28
smart pointers, 98
  and containers, 95
  and function parameters, 46
  and overuse, 25
  for resource management, 24
Socket, 74
sort, 18, 125, 166
spaces
  vs. tabs, 3
spaghetti, 17
special member functions. *See* default constructor; copy construction; copy assignment; destructor
specialization
  and not function templates, 126
  of class templates, not function templates, 127
  partial, 126
speculative execution, 16
Spencer, Henry, 173, 177
Square, 64
ssh, 8
stable_partition, 166

stable_sort, 166
stack unwinding, 92
standards, xi
  advantages of, xii
  what not to include, 2
Star Trek
  gratuitous reference to, 61
state
  global. *See* global variables
static
  misuse of, 112
static type checking, 120
static_cast, 181
  and not pointers, 178
  downcasting with, 29
statically checked errors. *See* errors, static checking
STL
  algorithms. *See* algorithms
  checked implementation valuable, 160
  containers. *See* containers
  iterators. *See* iterators
  searching, 165
  sorting, 166
  use leads to fewer explicit loops, 162
  using, 18
STL containers
  and thread safety, 21
string. *See* basic_string, *See* basic_string
String, 75
Stroustrup, Bjarne, xv, 32, 55, 119, 129, 149, 159
strtok, 54
style
  design vs. coding, 11
substitutability, 59, 64, 66
subsumption, 120
SummarizeFile, 116
super_string, 60
surprises
  programmers hate, 53
Sussman, Gerald Jay, 13
swap, 93, 100, 125, 126, 127
  never fails, 92
swap trick, 157
switch
  default case, 5

T

tabs
  vs. spaces, 3
taste
  matters of personal, 2
tautologies
  perfect for assertions, 131
template customization. *See* customization
Template Method, 68, 90
templates
  and implicit interface. *See* implicit interface
  and source-level dependencies, 42
  function. *See* function templates
  function templates not in same namespace as a type, 106
  macros interfering with, 33
  unintentionally nongeneric code, 128
temporaries
  avoid as constructor parameters, 13
temporary objects, 18, 51, 70, 98
Tensor, 47
terminate, 146
testing, 20
tests
  unit tests, 8
TeX
  The Errors of TeX, 11
this
  import, xv
thread safely, 21
thread safety, 21
  "just enough", 23
  and STL containers, 21
threads, 133
  and exceptions, 114
  multiple, 21
  vast majority of objects not shared across, 22
thrill sports, 152
time pressure, xiii
traits classes. *See* classes, traits
transform, 162
  example use of, 163
Translate, 117
Transmogrify, 54, 96
Transmogrify2, 97
Transubstantiate, 96
Tree, 25
TreeNode, 73
try, 38
two-phase lookup, 125
two's complement, 176
type checking
  static, 120
type safety, 28, 173, 176
type switching
  vs. polymorphism, 174
type system
  and not macros, 32
  and not memcpy/memcmp, 182
  hole in, 179
  making use of, 28, 29, 30, 131, 146, 173
type systems
  static vs. dynamic, 28
typename
  example use of, 122, 123, 125
types
  vs. representations, 176

U

unary_function, 91, 170, 172
Uncle Bob, xv
undefined behavior, 19, 25, 27, 36, 39, 61, 71, 88, 90, 91, 93, 173, 179, 181, 182, 183, 184, 185
unexpected_handler, 146
uninitialized variables, 36
unintentionally nongeneric code, 128
unions, 183

unit of work. *See* function
unit tests, 8
UnknownException, 146
unnamed namespace
  and not headers, 113
unqualified
  vs. qualified, 123
unsigned
  mismatch with signed. *See* signed/unsigned mismatch
unused parameters, 5
unwinding
  stack, 92
upper_bound, 15, 165
using, 83
  avoiding need for, 105
  is good, 108
  not before an #include, 108

V

validation
  of input data, 29
value-like types. *See* classes, value
Vandevoorde, Daveed, xv
varargs, 184
variable
  defined but not used, 5
  not initialized, 5
variable naming. *See* Hungarian notation
variables
  declaring, 35
  global. *See* global variables
  initialization of, 35
  initializing, 36
  uninitialized, 27
VCS. *See* version control system
vector. *See also* containers
  by default, 150
  insert, 139
  vs. list, 151
version control system, 8
versioning, 103, 138
  and get/set, 72
viral const, 30
virtual constructors, 88
virtual functions, 66
  and constructors and destructors, 88
  destructors, 90
  nonpublic preferred, 68
Visitor, 41, 121, 162, *See also* Acyclic Visitor
volatile, 37

W

Wagner, Luke, xv
warnings
  compiler. *See* compiler warnings
  disabling, 6
  none on successful build, 7
  spurious, dealing with, 6
Weinberg, Gerald, 1
what, 147
Wilson, Matthew, xv
works-like-a. *See*
  substitutability , *See*
  substitutability
wrapping
  header files. *See* header files, wrapping third-party headers
  platform-dependent operations, 21
Wysong, Lara, xv

Z

zero initialization, 39
Zolman, Leor, xv