O'REILLY®

Effective Modern C++

42 SPECIFIC WAYS TO IMPROVE YOUR USE OF C++11 AND C++14

Scott Meyers

O'REILLY®

Effective Modern C++

Coming to grips with C++11 and C++14 is more than a matter of familiarizing yourself with the features they introduce (e.g., `auto` type declarations, move semantics, lambda expressions, and concurrency support). The challenge is learning to use those features *effectively*—so that your software is correct, efficient, maintainable, and portable. That's where this practical book comes in. It describes how to write truly great software using C++11 and C++14—i.e., using *modern* C++.

Topics include:

- The pros and cons of braced initialization, `noexcept` specifications, perfect forwarding, and smart pointer `make` functions
- The relationships among `std::move`, `std::forward`, rvalue references, and universal references
- Techniques for writing clear, correct, *effective* lambda expressions
- How `std::atomic` differs from `volatile`, how each should be used, and how they relate to C++'s concurrency API
- How best practices in "old" C++ programming (i.e., C++98) require revision for software development in modern C++

Effective Modern C++ follows the proven guideline-based, example-driven format of Scott Meyers' earlier books, but covers entirely new material. It's essential reading for every modern C++ software developer.

"After I learned the C++ basics, I then learned how to use C++ in production code from Meyers' series of *Effective C++* books. *Effective Modern C++* is the most important how-to book for advice on key guidelines, styles, and idioms to use modern C++ effectively and well. Don't own it yet? Buy this one. Now."

—Herb Sutter
Chair of ISO C++ Standards Committee and C++ Software Architect at Microsoft

For more than 20 years, **Scott Meyers**' *Effective C++* books (*Effective C++*, *More Effective C++*, and *Effective STL*) have set the bar for C++ programming guidance. His clear, engaging explanations of complex technical material have earned him a worldwide following, keeping him in demand as a trainer, consultant, and conference presenter. He has a Ph.D. in Computer Science from Brown University.

PROGRAMMING/C++

US $49.99 **CAN $52.99**
ISBN: 978-1-491-90399-5

Twitter: @oreillymedia
facebook.com/oreilly

Praise for Effective Modern C++

So, still interested in C++? You should be! Modern C++ (i.e., C++11/C++14) is far more than just a facelift. Considering the new features, it seems that it's more a reinvention. Looking for guidelines and assistance? Then this book is surely what you are looking for. Concerning C++, Scott Meyers was and still is a synonym for accuracy, quality, and delight.

—Gerhard Kreuzer
Research and Development Engineer, Siemens AG

Finding utmost expertise is hard enough. Finding teaching perfectionism—an author's obsession with strategizing and streamlining explanations—is also difficult. You know you're in for a treat when you get to find both embodied in the same person. *Effective Modern C++* is a towering achievement from a consummate technical writer. It layers lucid, meaningful, and well-sequenced clarifications on top of complex and interconnected topics, all in crisp literary style. You're equally unlikely to find a technical mistake, a dull moment, or a lazy sentence in *Effective Modern C++*.

—Andrei Alexandrescu
Ph.D., Research Scientist, Facebook, and author of *Modern C++ Design*

As someone with over two decades of C++ experience, to get the most out of modern C++ (both best practices and pitfalls to avoid), I highly recommend getting this book, reading it thoroughly, and referring to it often! I've certainly learned new things going through it!

—Nevin Liber
Senior Software Engineer, DRW Trading Group

Bjarne Stroustrup—the creator of C++—said, "C++11 feels like a new language." *Effective Modern C++* makes us share this same feeling by clearly explaining how everyday programmers can benefit from new features and idioms of C++11 and C++14. Another great Scott Meyers book.

—Cassio Neri
FX Quantitative Analyst, Lloyds Banking Group

Scott has the knack of boiling technical complexity down to an understandable kernel. His *Effective C++* books helped to raise the coding style of a previous generation of C++ programmers; the new book seems positioned to do the same for those using modern C++.

—Roger Orr
OR/2 Limited, a member of the ISO C++ standards committee

Effective Modern C++ is a great tool to improve your modern C++ skills. Not only does it teach you how, when and where to use modern C++ and be effective, it also explains *why*. Without doubt, Scott's clear and insightful writing, spread over 42 well-thought items, gives programmers a much better understanding of the language.

—Bart Vandewoestyne
Research and Development Engineer and C++ enthusiast

I love C++, it has been my work vehicle for many decades now. And with the latest raft of features it is even more powerful and expressive than I would have previously imagined. But with all this choice comes the question "when and how do I apply these features?" As has always been the case, Scott's *Effective C++* books are the definitive answer to this question.

—Damien Watkins
Computation Software Engineering Team Lead, CSIRO

Great read for transitioning to modern C++—new C++11/14 language features are described alongside C++98, subject items are easy to reference, and advice summarized at the end of each section. Entertaining and useful for both casual and advanced C++ developers.

—Rachel Cheng
F5 Networks

If you're migrating from C++98/03 to C++11/14, you need the eminently practical and clear information Scott provides in Effective Modern C++. If you're already writing C++11 code, you'll probably discover issues with the new features through Scott's thorough discussion of the important new features of the language. Either way, this book is worth your time.

—Rob Stewart
Boost Steering Committee member (*boost.org*)

Effective Modern C++

Scott Meyers

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Effective Modern C++
by Scott Meyers

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Rachel Roumeliotis
Production Editor: Melanie Yarbrough
Copyeditor: Jasmine Kwityn
Proofreader: Charles Roumeliotis
Indexer: Scott Meyers
Interior Designer: David Futato
Cover Designer: Ellie Volkhausen
Illustrator: Rebecca Demarest

November 2014: First Edition

Revision History for the First Edition

2014-11-07:	First Release
2014-12-12:	Second Release
2015-01-20:	Third Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491903995* for release details.

978-1-491-90399-5

[V]

For Darla,

black Labrador Retriever extraordinaire

Table of Contents

From the Publisher xi

Acknowledgments xiii

Introduction 1

1. Deducing Types 9

Item 1: Understand template type deduction. 9
Item 2: Understand `auto` type deduction. 18
Item 3: Understand `decltype`. 23
Item 4: Know how to view deduced types. 30

2. `auto` 37

Item 5: Prefer `auto` to explicit type declarations. 37
Item 6: Use the explicitly typed initializer idiom when `auto` deduces undesired types. 43

3. Moving to Modern C++ 49

Item 7: Distinguish between `()` and `{}` when creating objects. 49
Item 8: Prefer `nullptr` to `0` and `NULL`. 58
Item 9: Prefer alias declarations to `typedef`s. 63
Item 10: Prefer scoped `enum`s to unscoped `enum`s. 67
Item 11: Prefer deleted functions to private undefined ones. 74
Item 12: Declare overriding functions `override`. 79
Item 13: Prefer `const_iterator`s to `iterator`s. 86
Item 14: Declare functions `noexcept` if they won't emit exceptions. 90
Item 15: Use `constexpr` whenever possible. 97

Item 16: Make `const` member functions thread safe. 103
Item 17: Understand special member function generation. 109

4. Smart Pointers. 117
Item 18: Use `std::unique_ptr` for exclusive-ownership resource management. 118
Item 19: Use `std::shared_ptr` for shared-ownership resource management. 125
Item 20: Use `std::weak_ptr` for `std::shared_ptr`-like pointers that can dangle. 134
Item 21: Prefer `std::make_unique` and `std::make_shared` to direct use of `new`. 139
Item 22: When using the Pimpl Idiom, define special member functions in the implementation file. 147

5. Rvalue References, Move Semantics, and Perfect Forwarding. 157
Item 23: Understand `std::move` and `std::forward`. 158
Item 24: Distinguish universal references from rvalue references. 164
Item 25: Use `std::move` on rvalue references, `std::forward` on universal references. 168
Item 26: Avoid overloading on universal references. 177
Item 27: Familiarize yourself with alternatives to overloading on universal references. 184
Item 28: Understand reference collapsing. 197
Item 29: Assume that move operations are not present, not cheap, and not used. 203
Item 30: Familiarize yourself with perfect forwarding failure cases. 207

6. Lambda Expressions. 215
Item 31: Avoid default capture modes. 216
Item 32: Use init capture to move objects into closures. 224
Item 33: Use `decltype` on `auto&&` parameters to `std::forward` them. 229
Item 34: Prefer lambdas to `std::bind`. 232

7. The Concurrency API. 241
Item 35: Prefer task-based programming to thread-based. 241
Item 36: Specify `std::launch::async` if asynchronicity is essential. 245
Item 37: Make `std::threads` unjoinable on all paths. 250
Item 38: Be aware of varying thread handle destructor behavior. 258
Item 39: Consider `void` futures for one-shot event communication. 262

Item 40: Use `std::atomic` for concurrency, `volatile` for special memory. 271

8. Tweaks. 281

Item 41: Consider pass by value for copyable parameters that are cheap to move and always copied. 281

Item 42: Consider emplacement instead of insertion. 292

Index. 303

From the Publisher

Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Effective Modern C++* by Scott Meyers (O'Reilly). Copyright 2015 Scott Meyers, 978-1-491-90399-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Comments and questions concerning this book may be addressed to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I started investigating what was then known as C++0x (the nascent C++11) in 2009. I posted numerous questions to the Usenet newsgroup `comp.std.c++`, and I'm grateful to the members of that community (especially Daniel Krügler) for their very helpful postings. In more recent years, I've turned to Stack Overflow when I had questions about C++11 and C++14, and I'm equally indebted to that community for its help in understanding the finer points of modern C++.

In 2010, I prepared materials for a training course on C++0x (ultimately published as *Overview of the New C++*, Artima Publishing, 2010). Both those materials and my knowledge greatly benefited from the technical vetting performed by Stephan T. Lavavej, Bernhard Merkle, Stanley Friesen, Leor Zolman, Hendrik Schober, and Anthony Williams. Without their help, I would probably never have been in a position to undertake *Effective Modern C++*. That title, incidentally, was suggested or endorsed by several readers responding to my 18 February 2014 blog post, "Help me name my book," and Andrei Alexandrescu (author of *Modern C++ Design*, Addison-Wesley, 2001) was kind enough to bless the title as not poaching on his terminological turf.

I'm unable to identify the origins of all the information in this book, but some sources had a relatively direct impact. Item 4's use of an undefined template to coax type information out of compilers was suggested by Stephan T. Lavavej, and Matt P. Dziubinski brought Boost.TypeIndex to my attention. In Item 5, the `unsigned`-`std::vector<int>::size_type` example is from Andrey Karpov's 28 February 2010 article, "In what way can C++0x standard help you eliminate 64-bit errors." The `std::pair<std::string, int>`/`std::pair<const std::string, int>` example in the same Item is from Stephan T. Lavavej's talk at *Going Native 2012*, "STL11: Magic && Secrets." Item 6 was inspired by Herb Sutter's 12 August 2013 article, "GotW #94 Solution: AAA Style (Almost Always Auto)." Item 9 was motivated by Martinho Fernandes' blog post of 27 May 2012, "Handling dependent names." The Item 12 example demonstrating overloading on reference qualifiers is based on Casey's answer to the question, "What's a use case for overloading member functions on reference qualifiers?," posted to Stack Overflow on 14 January 2014. My Item 15 treatment of C++14's expanded support for `constexpr` functions incorporates information I received from Rein Halbersma. Item 16 is based on Herb Sutter's *C++ and Beyond 2012* presentation, "You don't know `const` and `mutable`." Item 18's advice to have factory functions return `std::unique_ptr`s is based on Herb Sutter's 30 May 2013 article, "GotW# 90 Solution: Factories." In Item 19, `fastLoadWidget` is derived from Herb Sutter's *Going Native 2013* presentation, "My Favorite C++ 10-Liner." My treatment of `std::unique_ptr` and incomplete types in Item 22 draws on Herb Sutter's 27 November 2011 article, "GotW #100: Compilation Firewalls" as well as Howard Hinnant's 22 May 2011 answer to the Stack Overflow question, "Is std::unique_ptr<T> required to know the full definition of T?" The `Matrix` addition example in Item 25 is based on writings by David Abrahams. JoeArgonne's 8 December 2012 comment on the 30 November 2012 blog post, "Another alternative to lambda move capture," was the source of Item 32's `std::bind`-based approach to emulating init capture in C++11. Item 37's explanation of the problem with an implicit detach in `std::thread`'s destructor is taken from Hans-J. Boehm's 4 December 2008 paper, "N2802: A plea to reconsider detach-on-destruction for thread objects." Item 41 was originally motivated by discussions of David Abrahams' 15 August 2009 blog post, "Want speed? Pass by value." The idea that move-only types deserve special treatment is due to Matthew Fioravante, while the analysis of assignment-based copying stems from comments by Howard Hinnant. In Item 42, Stephan T. Lavavej and Howard Hinnant helped me understand the relative performance profiles of emplacement and insertion functions, and Michael Winterberg brought to my attention how emplacement can lead to resource leaks. (Michael credits Sean Parent's *Going Native 2013* presentation, "C++ Seasoning," as his source). Michael also pointed out how emplacement functions use direct initialization, while insertion functions use copy initialization.

Reviewing drafts of a technical book is a demanding, time-consuming, and utterly critical task, and I'm fortunate that so many people were willing to do it for me. Full or partial drafts of *Effective Modern C++* were officially reviewed by Cassio Neri, Nate Kohl, Gerhard Kreuzer, Leor Zolman, Bart Vandewoestyne, Stephan T. Lavavej, Nevin ":-)" Liber, Rachel Cheng, Rob Stewart, Bob Steagall, Damien Watkins, Bradley E. Needham, Rainer Grimm, Fredrik Winkler, Jonathan Wakely, Herb Sutter, Andrei Alexandrescu, Eric Niebler, Thomas Becker, Roger Orr, Anthony Williams, Michael Winterberg, Benjamin Huchley, Tom Kirby-Green, Alexey A Nikitin, William Dealtry, Hubert Matthews, and Tomasz Kamiński. I also received feedback from several readers through O'Reilly's Early Release EBooks and Safari Books Online's Rough Cuts, comments on my blog (*The View from Aristeia*), and email. I'm grateful to each of these people. The book is *much* better than it would have been without their help. I'm particularly indebted to Stephan T. Lavavej and Rob Stewart, whose extraordinarily detailed and comprehensive remarks lead me to worry that they spent nearly as much time on this book as I did. Special thanks also go to Leor Zolman, who, in addition to reviewing the manuscript, double-checked all the code examples.

Dedicated reviews of digital versions of the book were performed by Gerhard Kreuzer, Emyr Williams, and Bradley E. Needham.

My decision to limit the line length in code displays to 64 characters (the maximum likely to display properly in print as well as across a variety of digital devices, device orientations, and font configurations) was based on data provided by Michael Maher.

Since initial publication, I've incorporated bug fixes and other improvements suggested by Kostas Vlahavas, Daniel Alonso Alemany, Takatoshi Kondo, Bartek Szurgot, Tyler Brock, Jay Zipnick, Barry Revzin, Robert McCabe, Oliver Bruns, Fabrice Ferino, Dainis Jonitis, Petr Valasek, and Bart Vandewoestyne. Many thanks to all these people for helping improve the accuracy and clarity of *Effective Modern C++*.

Ashley Morgan Williams made dining at the Lake Oswego Pizzicato uniquely entertaining. When it comes to man-sized Caesars, she's the go-to gal.

More than 20 years after first living through my playing author, my wife, Nancy L. Urbano, once again tolerated many months of distracted conversations with a cocktail of resignation, exasperation, and timely splashes of understanding and support. During the same period, our dog, Darla, was largely content to doze away the hours I spent staring at computer screens, but she never let me forget that there's life beyond the keyboard.

Introduction

If you're an experienced C++ programmer and are anything like me, you initially approached C++11 thinking, "Yes, yes, I get it. It's C++, only more so." But as you learned more, you were surprised by the scope of the changes. `auto` declarations, range-based `for` loops, lambda expressions, and rvalue references change the face of C++, to say nothing of the new concurrency features. And then there are the idiomatic changes. `0` and `typedef`s are out, `nullptr` and alias declarations are in. Enums should now be scoped. Smart pointers are now preferable to built-in ones. Moving objects is normally better than copying them.

There's a lot to learn about C++11, not to mention C++14.

More importantly, there's a lot to learn about making *effective* use of the new capabilities. If you need basic information about "modern" C++ features, resources abound, but if you're looking for guidance on how to employ the features to create software that's correct, efficient, maintainable, and portable, the search is more challenging. That's where this book comes in. It's devoted not to describing the features of C++11 and C++14, but instead to their effective application.

The information in the book is broken into guidelines called *Items*. Want to understand the various forms of type deduction? Or know when (and when not) to use `auto` declarations? Are you interested in why `const` member functions should be thread safe, how to implement the Pimpl Idiom using `std::unique_ptr`, why you should avoid default capture modes in lambda expressions, or the differences between `std::atomic` and `volatile`? The answers are all here. Furthermore, they're platform-independent, Standards-conformant answers. This is a book about *portable* C++.

The Items in this book are guidelines, not rules, because guidelines have exceptions. The most important part of each Item is not the advice it offers, but the rationale behind the advice. Once you've read that, you'll be in a position to determine whether the circumstances of your project justify a violation of the Item's guidance. The true goal of this book isn't to tell you what to do or what to avoid doing, but to convey a deeper understanding of how things work in C++11 and C++14.

Terminology and Conventions

To make sure we understand one another, it's important to agree on some terminology, beginning, ironically, with "C++." There have been four official versions of C++, each named after the year in which the corresponding ISO Standard was adopted: *C++98*, *C++03*, *C++11*, and *C++14*. C++98 and C++03 differ only in technical details, so in this book, I refer to both as C++98. When I refer to C++11, I mean both C++11 and C++14, because C++14 is effectively a superset of C++11. When I write C++14, I mean specifically C++14. And if I simply mention C++, I'm making a broad statement that pertains to all language versions.

Term I Use	Language Versions I Mean
C++	All
C++98	C++98 and C++03
C++11	C++11 and C++14
C++14	C++14

As a result, I might say that C++ places a premium on efficiency (true for all versions), that C++98 lacks support for concurrency (true only for C++98 and C++03), that C++11 supports lambda expressions (true for C++11 and C++14), and that C++14 offers generalized function return type deduction (true for C++14 only).

C++11's most pervasive feature is probably move semantics, and the foundation of move semantics is distinguishing expressions that are *rvalues* from those that are *lvalues*. That's because rvalues indicate objects eligible for move operations, while lvalues generally don't. In concept (though not always in practice), rvalues correspond to temporary objects returned from functions, while lvalues correspond to objects you can refer to, either by name or by following a pointer or lvalue reference.

A useful heuristic to determine whether an expression is an lvalue is to ask if you can take its address. If you can, it typically is. If you can't, it's usually an rvalue. A nice feature of this heuristic is that it helps you remember that the type of an expression is independent of whether the expression is an lvalue or an rvalue. That is, given a type `T`, you can have lvalues of type `T` as well as rvalues of type `T`. It's especially important to remember this when dealing with a parameter of rvalue reference type, because the parameter itself is an lvalue:

```
class Widget {
public:
  Widget(Widget&& rhs);    // rhs is an lvalue, though it has
  …                        // an rvalue reference type
};
```

Here, it'd be perfectly valid to take `rhs`'s address inside `Widget`'s move constructor, so `rhs` is an lvalue, even though its type is an rvalue reference. (By similar reasoning, all parameters are lvalues.)

That code snippet demonstrates several conventions I normally follow:

- The class name is `Widget`. I use `Widget` whenever I want to refer to an arbitrary user-defined type. Unless I need to show specific details of the class, I use `Widget` without declaring it.
- I use the parameter name *`rhs`* ("right-hand side"). It's my preferred parameter name for the *move operations* (i.e., move constructor and move assignment operator) and the *copy operations* (i.e., copy constructor and copy assignment operator). I also employ it for the right-hand parameter of binary operators:

  ```
  Matrix operator+(const Matrix& lhs, const Matrix& rhs);
  ```

  It's no surprise, I hope, that *`lhs`* stands for "left-hand side."
- I apply special formatting to parts of code or parts of comments to draw your attention to them. In the `Widget` move constructor above, I've highlighted the declaration of `rhs` and the part of the comment noting that `rhs` is an lvalue. Highlighted code is neither inherently good nor inherently bad. It's simply code you should pay particular attention to.
- I use "…" to indicate "other code could go here." This narrow ellipsis is different from the wide ellipsis ("`...`") that's used in the source code for C++11's variadic templates. That sounds confusing, but it's not. For example:

  ```
  template<typename... Ts>                // these are C++
  void processVals(const Ts&... params)   // source code
  {                                       // ellipses

    …                                     // this means "some
                                          // code goes here"
  }
  ```

  The declaration of `processVals` shows that I use `typename` when declaring type parameters in templates, but that's merely a personal preference; the keyword `class` would work just as well. On those occasions where I show code excerpts from a C++ Standard, I declare type parameters using `class`, because that's what the Standards do.

When an object is initialized with another object of the same type, the new object is said to be a *copy* of the initializing object, even if the copy was created via the move constructor. Regrettably, there's no terminology in C++ that distinguishes between an object that's a copy-constructed copy and one that's a move-constructed copy:

```
void someFunc(Widget w);        // someFunc's parameter w
                                // is passed by value

Widget wid;                     // wid is some Widget

someFunc(wid);                  // in this call to someFunc,
                                // w is a copy of wid that's
                                // created via copy construction

someFunc(std::move(wid));       // in this call to SomeFunc,
                                // w is a copy of wid that's
                                // created via move construction
```

Copies of rvalues are generally move constructed, while copies of lvalues are usually copy constructed. An implication is that if you know only that an object is a copy of another object, it's not possible to say how expensive it was to construct the copy. In the code above, for example, there's no way to say how expensive it is to create the parameter `w` without knowing whether rvalues or lvalues are passed to `someFunc`. (You'd also have to know the cost of moving and copying `Widget`s.)

In a function call, the expressions passed at the call site are the function's *arguments*. The arguments are used to initialize the function's *parameters*. In the first call to `someFunc` above, the argument is `wid`. In the second call, the argument is `std::move(wid)`. In both calls, the parameter is `w`. The distinction between arguments and parameters is important, because parameters are lvalues, but the arguments with which they are initialized may be rvalues or lvalues. This is especially relevant during the process of *perfect forwarding*, whereby an argument passed to a function is passed to a second function such that the original argument's rvalueness or lvalueness is preserved. (Perfect forwarding is discussed in detail in Item 30.)

Well-designed functions are *exception safe*, meaning they offer at least the basic exception safety guarantee (i.e., the *basic guarantee*). Such functions assure callers that even if an exception is thrown, program invariants remain intact (i.e., no data structures are corrupted) and no resources are leaked. Functions offering the strong exception safety guarantee (i.e., the *strong guarantee*) assure callers that if an exception arises, the state of the program remains as it was prior to the call.

When I refer to a *function object*, I usually mean an object of a type supporting an `operator()` member function. In other words, an object that acts like a function. Occasionally I use the term in a slightly more general sense to mean anything that can be invoked using the syntax of a non-member function call (i.e., "*`functionName(arguments)`*"). This broader definition covers not just objects supporting `operator()`, but also functions and C-like function pointers. (The narrower definition comes from C++98, the broader one from C++11.) Generalizing further by adding member function pointers yields what are known as *callable objects*. You can generally ignore the fine distinctions and simply think of function objects and callable objects as things in C++ that can be invoked using some kind of function-calling syntax.

Function objects created through lambda expressions are known as *closures*. It's seldom necessary to distinguish between lambda expressions and the closures they create, so I often refer to both as *lambdas*. Similarly, I rarely distinguish between *function templates* (i.e., templates that generate functions) and *template functions* (i.e., the functions generated from function templates). Ditto for *class templates* and *template classes*.

Many things in C++ can be both declared and defined. *Declarations* introduce names and types without giving details, such as where storage is located or how things are implemented:

```
extern int x;                    // object declaration

class Widget;                    // class declaration

bool func(const Widget& w);      // function declaration

enum class Color;                // scoped enum declaration
                                 // (see Item 10)
```

Definitions provide the storage locations or implementation details:

```
int x;                           // object definition

class Widget {                   // class definition
  …
};

bool func(const Widget& w)
{ return w.size() < 10; }        // function definition

enum class Color
{ Yellow, Red, Blue };           // scoped enum definition
```

A definition also qualifies as a declaration, so unless it's really important that something is a definition, I tend to refer to declarations.

I define a function's *signature* to be the part of its declaration that specifies parameter and return types. Function and parameter names are not part of the signature. In the example above, `func`'s signature is `bool(const Widget&)`. Elements of a function's declaration other than its parameter and return types (e.g., `noexcept` or `constexpr`, if present), are excluded. (`noexcept` and `constexpr` are described in Items 14 and 15.) The official definition of "signature" is slightly different from mine, but for this book, my definition is more useful. (The official definition sometimes omits return types.)

New C++ Standards generally preserve the validity of code written under older ones, but occasionally the Standardization Committee *deprecates* features. Such features are on standardization death row and may be removed from future Standards. Compilers may or may not warn about the use of deprecated features, but you should do your best to avoid them. Not only can they lead to future porting headaches, they're generally inferior to the features that replace them. For example, `std::auto_ptr` is deprecated in C++11, because `std::unique_ptr` does the same job, only better.

Sometimes a Standard says that the result of an operation is *undefined behavior*. That means that runtime behavior is unpredictable, and it should go without saying that you want to steer clear of such uncertainty. Examples of actions with undefined behavior include using square brackets ("`[]`") to index beyond the bounds of a `std::vector`, dereferencing an uninitialized iterator, or engaging in a data race (i.e., having two or more threads, at least one of which is a writer, simultaneously access the same memory location).

I call built-in pointers, such as those returned from `new`, *raw pointers*. The opposite of a raw pointer is a *smart pointer*. Smart pointers normally overload the pointer-dereferencing operators (`operator->` and `operator*`), though Item 20 explains that `std::weak_ptr` is an exception.

In source code comments, I sometimes abbreviate "constructor" as *`ctor`* and "destructor" as *`dtor`*.

Reporting Bugs and Suggesting Improvements

I've done my best to fill this book with clear, accurate, useful information, but surely there are ways to make it better. If you find errors of any kind (technical, expository, grammatical, typographical, etc.), or if you have suggestions for how the book could be improved, please email me at *emc++@aristeia.com*. New printings give me the