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

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

Printed in the Canada.

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

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

978-1-491-90399-5

[TI]

For Darla,

black Labrador Retriever extraordinaire

Table of Contents

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*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

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_ptrs` 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 reviwing 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.

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 opportunity to revise *Effective Modern C++*, and I can't address issues I don't know about!

To view the list of the issues I do know about, consult the book's errata page, *http://www.aristeia.com/BookErrata/emc++-errata.html*.

CHAPTER 1
Deducing Types

C++98 had a single set of rules for type deduction: the one for function templates. C++11 modifies that ruleset a bit and adds two more, one for `auto` and one for `decltype`. C++14 then extends the usage contexts in which `auto` and `decltype` may be employed. The increasingly widespread application of type deduction frees you from the tyranny of spelling out types that are obvious or redundant. It makes C++ software more adaptable, because changing a type at one point in the source code automatically propagates through type deduction to other locations. However, it can render code more difficult to reason about, because the types deduced by compilers may not be as apparent as you'd like.

Without a solid understanding of how type deduction operates, effective programming in modern C++ is all but impossible. There are just too many contexts where type deduction takes place: in calls to function templates, in most situations where `auto` appears, in `decltype` expressions, and, as of C++14, where the enigmatic `decltype(auto)` construct is employed.

This chapter provides the information about type deduction that every C++ developer requires. It explains how template type deduction works, how `auto` builds on that, and how `decltype` goes its own way. It even explains how you can force compilers to make the results of their type deductions visible, thus enabling you to ensure that compilers are deducing the types you want them to.

Item 1: Understand template type deduction.

When users of a complex system are ignorant of how it works, yet happy with what it does, that says a lot about the design of the system. By this measure, template type deduction in C++ is a tremendous success. Millions of programmers have passed arguments to template functions with completely satisfactory results, even though many of those programmers would be hard-pressed to give more than the haziest description of how the types used by those functions were deduced.

If that group includes you, I have good news and bad news. The good news is that type deduction for templates is the basis for one of modern C++'s most compelling features: `auto`. If you were happy with how C++98 deduced types for templates, you're set up to be happy with how C++11 deduces types for `auto`. The bad news is that when the template type deduction rules are applied in the context of `auto`, they sometimes seem less intuitive than when they're applied to templates. For that reason, it's important to truly understand the aspects of template type deduction that `auto` builds on. This Item covers what you need to know.

If you're willing to overlook a pinch of pseudocode, we can think of a function template as looking like this:

```
template<typename T>
void f(ParamType param);
```

A call can look like this:

```
f(expr);                        // call f with some expression
```

During compilation, compilers use *`expr`* to deduce two types: one for `T` and one for *`ParamType`*. These types are frequently different, because *`ParamType`* often contains adornments, e.g., `const` or reference qualifiers. For example, if the template is declared like this,

```
template<typename T>
void f(const T& param);         // ParamType is const T&
```

and we have this call,

```
int x = 0;

f(x);                           // call f with an int
```

`T` is deduced to be `int`, but *`ParamType`* is deduced to be `const int&`.

It's natural to expect that the type deduced for `T` is the same as the type of the argument passed to the function, i.e., that `T` is the type of *`expr`*. In the above example, that's the case: `x` is an `int`, and `T` is deduced to be `int`. But it doesn't always work that way. The type deduced for `T` is dependent not just on the type of *`expr`*, but also on the form of *`ParamType`*. There are three cases:

- *ParamType* is a pointer or reference type, but not a universal reference. (Universal references are described in Item 24. At this point, all you need to know is that they exist and that they're not the same as lvalue references or rvalue references.)
- *ParamType* is a universal reference.
- *ParamType* is neither a pointer nor a reference.

We therefore have three type deduction scenarios to examine. Each will be based on our general form for templates and calls to it:

```
template<typename T>
void f(ParamType param);

f(expr);                  // deduce T and ParamType from expr
```

Case 1: *ParamType* is a Reference or Pointer, but not a Universal Reference

The simplest situation is when *ParamType* is a reference type or a pointer type, but not a universal reference. In that case, type deduction works like this:

1. If *expr*'s type is a reference, ignore the reference part.
2. Then pattern-match *expr*'s type against *ParamType* to determine `T`.

For example, if this is our template,

```
template<typename T>
void f(T& param);         // param is a reference
```

and we have these variable declarations,

```
int x = 27;               // x is an int
const int cx = x;         // cx is a const int
const int& rx = x;        // rx is a reference to x as a const int
```

the deduced types for `param` and `T` in various calls are as follows:

```
f(x);                     // T is int, param's type is int&

f(cx);                    // T is const int,
                          // param's type is const int&

f(rx);                    // T is const int,
                          // param's type is const int&
```

In the second and third calls, notice that because cx and rx designate const values, T is deduced to be const int, thus yielding a parameter type of const int&. That's important to callers. When they pass a const object to a reference parameter, they expect that object to remain unmodifiable, i.e., for the parameter to be a reference-to-const. That's why passing a const object to a template taking a T& parameter is safe: the constness of the object becomes part of the type deduced for T.

In the third example, note that even though rx's type is a reference, T is deduced to be a non-reference. That's because rx's reference-ness is ignored during type deduction.

These examples all show lvalue reference parameters, but type deduction works exactly the same way for rvalue reference parameters. Of course, only rvalue arguments may be passed to rvalue reference parameters, but that restriction has nothing to do with type deduction.

If we change the type of f's parameter from T& to const T&, things change a little, but not in any really surprising ways. The constness of cx and rx continues to be respected, but because we're now assuming that param is a reference-to-const, there's no longer a need for const to be deduced as part of T:

```
template<typename T>
void f(const T& param);  // param is now a ref-to-const

int x = 27;              // as before
const int cx = x;        // as before
const int& rx = x;       // as before

f(x);                    // T is int, param's type is const int&

f(cx);                   // T is int, param's type is const int&

f(rx);                   // T is int, param's type is const int&
```

As before, rx's reference-ness is ignored during type deduction.

If param were a pointer (or a pointer to const) instead of a reference, things would work essentially the same way:

```
template<typename T>
void f(T* param);        // param is now a pointer

int x = 27;              // as before
const int *px = &x;      // px is a ptr to x as a const int
```

```
f(&x);                    // T is int, param's type is int*

f(px);                    // T is const int,
                          // param's type is const int*
```

By now, you may find yourself yawning and nodding off, because C++'s type deduction rules work so naturally for reference and pointer parameters, seeing them in written form is really dull. Everything's just obvious! Which is exactly what you want in a type deduction system.

Case 2: *ParamType* is a Universal Reference

Things are less obvious for templates taking universal reference parameters. Such parameters are declared like rvalue references (i.e., in a function template taking a type parameter `T`, a universal reference's declared type is `T&&`), but they behave differently when lvalue arguments are passed in. The complete story is told in Item 24, but here's the headline version:

- If *`expr`* is an lvalue, both `T` and *`ParamType`* are deduced to be lvalue references. That's doubly unusual. First, it's the only situation in template type deduction where `T` is deduced to be a reference. Second, although *`ParamType`* is declared using the syntax for an rvalue reference, its deduced type is an lvalue reference.
- If *`expr`* is an rvalue, the "normal" (i.e., Case 1) rules apply.

For example:

```
template<typename T>
void f(T&& param);        // param is now a universal reference

int x = 27;               // as before
const int cx = x;         // as before
const int& rx = x;        // as before

f(x);                     // x is lvalue, so T is int&,
                          // param's type is also int&

f(cx);                    // cx is lvalue, so T is const int&,
                          // param's type is also const int&

f(rx);                    // rx is lvalue, so T is const int&,
                          // param's type is also const int&

f(27);                    // 27 is rvalue, so T is int,
                          // param's type is therefore int&&
```

Item 24 explains exactly why these examples play out the way they do. The key point here is that the type deduction rules for universal reference parameters are different from those for parameters that are lvalue references or rvalue references. In particular, when universal references are in use, type deduction distinguishes between lvalue arguments and rvalue arguments. That never happens for non-universal references.

Case 3: *ParamType* is Neither a Pointer nor a Reference

When *ParamType* is neither a pointer nor a reference, we're dealing with pass-by-value:

```
template<typename T>
void f(T param);          // param is now passed by value
```

That means that `param` will be a copy of whatever is passed in—a completely new object. The fact that `param` will be a new object motivates the rules that govern how `T` is deduced from *expr*:

1. As before, if *expr*'s type is a reference, ignore the reference part.
2. If, after ignoring *expr*'s reference-ness, *expr* is `const`, ignore that, too. If it's `volatile`, also ignore that. (`volatile` objects are uncommon. They're generally used only for implementing device drivers. For details, see Item 40.)

Hence:

```
int x = 27;            // as before
const int cx = x;      // as before
const int& rx = x;     // as before

f(x);                  // T's and param's types are both int

f(cx);                 // T's and param's types are again both int

f(rx);                 // T's and param's types are still both int
```

Note that even though `cx` and `rx` represent `const` values, `param` isn't `const`. That makes sense. `param` is an object that's completely independent of `cx` and `rx`—a *copy* of `cx` or `rx`. The fact that `cx` and `rx` can't be modified says nothing about whether `param` can be. That's why *expr*'s `const`ness (and `volatile`ness, if any) is ignored when deducing a type for `param`: just because *expr* can't be modified doesn't mean that a copy of it can't be.

It's important to recognize that `const` (and `volatile`) is ignored only for by-value parameters. As we've seen, for parameters that are references-to- or pointers-to-`const`, the `const`ness of *expr* is preserved during type deduction. But consider the case where *expr* is a const pointer to a const object, and *expr* is passed to a by-value param:

```
template<typename T>
void f(T param);          // param is still passed by value

const char* const ptr =   // ptr is const pointer to const object
  "Fun with pointers";

f(ptr);                   // pass arg of type const char * const
```

Here, the const to the right of the asterisk declares ptr to be const: ptr can't be made to point to a different location, nor can it be set to null. (The const to the left of the asterisk says that what ptr points to—the character string—is const, hence can't be modified.) When ptr is passed to f, the bits making up the pointer are copied into param. As such, *the pointer itself (ptr) will be passed by value*. In accord with the type deduction rule for by-value parameters, the constness of ptr will be ignored, and the type deduced for param will be const char*, i.e., a modifiable pointer to a const character string. The constness of what ptr points to is preserved during type deduction, but the constness of ptr itself is ignored when copying it to create the new pointer, param.

Array Arguments

That pretty much covers it for mainstream template type deduction, but there's a niche case that's worth knowing about. It's that array types are different from pointer types, even though they sometimes seem to be interchangeable. A primary contributor to this illusion is that, in many contexts, an array *decays* into a pointer to its first element. This decay is what permits code like this to compile:

```
const char name[] = "J. P. Briggs";  // name's type is
                                     // const char[13]

const char * ptrToName = name;       // array decays to pointer
```

Here, the const char* pointer ptrToName is being initialized with name, which is a const char[13]. These types (const char* and const char[13]) are not the same, but because of the array-to-pointer decay rule, the code compiles.

But what if an array is passed to a template taking a by-value parameter? What happens then?

```
template<typename T>
void f(T param);       // template with by-value parameter
```

```
f(name);                // what types are deduced for T and param?
```

We begin with the observation that there is no such thing as a function parameter that's an array. Yes, yes, the syntax is legal,

```
void myFunc(int param[]);
```

but the array declaration is treated as a pointer declaration, meaning that `myFunc` could equivalently be declared like this:

```
void myFunc(int* param);         // same function as above
```

This equivalence of array and pointer parameters is a bit of foliage springing from the C roots at the base of C++, and it fosters the illusion that array and pointer types are the same.

Because array parameter declarations are treated as if they were pointer parameters, the type of an array that's passed to a template function by value is deduced to be a pointer type. That means that in the call to the template `f`, its type parameter `T` is deduced to be `const char*`:

```
f(name);            // name is array, but T deduced as const char*
```

But now comes a curve ball. Although functions can't declare parameters that are truly arrays, they *can* declare parameters that are *references* to arrays! So if we modify the template `f` to take its argument by reference,

```
template<typename T>
void f(T& param);      // template with by-reference parameter
```

and we pass an array to it,

```
f(name);               // pass array to f
```

the type deduced for `T` is the actual type of the array! That type includes the size of the array, so in this example, `T` is deduced to be `const char [13]`, and the type of `f`'s parameter (a reference to this array) is `const char (&)[13]`. Yes, the syntax looks toxic, but knowing it will score you mondo points with those few souls who care.

Interestingly, the ability to declare references to arrays enables creation of a template that deduces the number of elements that an array contains:

```
// return size of an array as a compile-time constant. (The
// array parameter has no name, because we care only about
// the number of elements it contains.)
template<typename T, std::size_t N>                 // see info
constexpr std::size_t arraySize(T (&)[N]) noexcept  // below on
{                                                   // constexpr
```

```
    return N;                                      // and
  }                                                // noexcept
```

As Item 15 explains, declaring this function `constexpr` makes its result available during compilation. That makes it possible to declare, say, an array with the same number of elements as a second array whose size is computed from a braced initializer:

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };     // keyVals has
                                                // 7 elements

int mappedVals[arraySize(keyVals)];             // so does
                                                // mappedVals
```

Of course, as a modern C++ developer, you'd naturally prefer a `std::array` to a built-in array:

```
std::array<int, arraySize(keyVals)> mappedVals;  // mappedVals'
                                                 // size is 7
```

As for `arraySize` being declared `noexcept`, that's to help compilers generate better code. For details, see Item 14.

Function Arguments

Arrays aren't the only things in C++ that can decay into pointers. Function types can decay into function pointers, and everything we've discussed regarding type deduction for arrays applies to type deduction for functions and their decay into function pointers. As a result:

```
void someFunc(int, double);   // someFunc is a function;
                              // type is void(int, double)

template<typename T>
void f1(T param);             // in f1, param passed by value

template<typename T>
void f2(T& param);            // in f2, param passed by ref

f1(someFunc);                 // param deduced as ptr-to-func;
                              // type is void (*)(int, double)

f2(someFunc);                 // param deduced as ref-to-func;
                              // type is void (&)(int, double)
```

This rarely makes any difference in practice, but if you're going to know about array-to-pointer decay, you might as well know about function-to-pointer decay, too.

So there you have it: the `auto`-related rules for template type deduction. I remarked at the outset that they're pretty straightforward, and for the most part, they are. The special treatment accorded lvalues when deducing types for universal references muddies the water a bit, however, and the decay-to-pointer rules for arrays and functions stirs up even greater turbidity. Sometimes you simply want to grab your compilers and demand, "Tell me what type you're deducing!" When that happens, turn to Item 4, because it's devoted to coaxing compilers into doing just that.

Things to Remember

- During template type deduction, arguments that are references are treated as non-references, i.e., their reference-ness is ignored.
- When deducing types for universal reference parameters, lvalue arguments get special treatment.
- When deducing types for by-value parameters, `const` and/or `volatile` arguments are treated as non-`const` and non-`volatile`.
- During template type deduction, arguments that are array or function names decay to pointers, unless they're used to initialize references.

Item 2: Understand `auto` type deduction.

If you've read Item 1 on template type deduction, you already know almost everything you need to know about `auto` type deduction, because, with only one curious exception, `auto` type deduction *is* template type deduction. But how can that be? Template type deduction involves templates and functions and parameters, but `auto` deals with none of those things.

That's true, but it doesn't matter. There's a direct mapping between template type deduction and `auto` type deduction. There is literally an algorithmic transformation from one to the other.

In Item 1, template type deduction is explained using this general function template

```
template<typename T>
void f(ParamType param);
```

and this general call:

```
f(expr);                    // call f with some expression
```

In the call to `f`, compilers use *`expr`* to deduce types for `T` and *`ParamType`*.

When a variable is declared using `auto`, `auto` plays the role of `T` in the template, and the type specifier for the variable acts as *`ParamType`*. This is easier to show than to describe, so consider this example:

```
auto x = 27;
```

Here, the type specifier for `x` is simply `auto` by itself. On the other hand, in this declaration,

```
const auto cx = x;
```

the type specifier is `const auto`. And here,

```
const auto& rx = x;
```

the type specifier is `const auto&`. To deduce types for `x`, `cx`, and `rx` in these examples, compilers act as if there were a template for each declaration as well as a call to that template with the corresponding initializing expression:

```
template<typename T>              // conceptual template for
void func_for_x(T param);         // deducing x's type

func_for_x(27);                   // conceptual call: param's
                                  // deduced type is x's type

template<typename T>              // conceptual template for
void func_for_cx(const T param);  // deducing cx's type

func_for_cx(x);                   // conceptual call: param's
                                  // deduced type is cx's type

template<typename T>              // conceptual template for
void func_for_rx(const T& param); // deducing rx's type

func_for_rx(x);                   // conceptual call: param's
                                  // deduced type is rx's type
```

As I said, deducing types for `auto` is, with only one exception (which we'll discuss soon), the same as deducing types for templates.

Item 1 divides template type deduction into three cases, based on the characteristics of *`ParamType`*, the type specifier for `param` in the general function template. In a variable declaration using `auto`, the type specifier takes the place of *`ParamType`*, so there are three cases for that, too:

- Case 1: The type specifier is a pointer or reference, but not a universal reference.
- Case 2: The type specifier is a universal reference.

- Case 3: The type specifier is neither a pointer nor a reference.

We've already seen examples of cases 1 and 3:

```
auto x = 27;          // case 3 (x is neither ptr nor reference)

const auto cx = x;    // case 3 (cx isn't either)

const auto& rx = x;   // case 1 (rx is a non-universal ref.)
```

Case 2 works as you'd expect:

```
auto&& uref1 = x;     // x is int and lvalue,
                      // so uref1's type is int&

auto&& uref2 = cx;    // cx is const int and lvalue,
                      // so uref2's type is const int&

auto&& uref3 = 27;    // 27 is int and rvalue,
                      // so uref3's type is int&&
```

Item 1 concludes with a discussion of how array and function names decay into pointers for non-reference type specifiers. That happens in `auto` type deduction, too:

```
const char name[] =            // name's type is const char[13]
  "R. N. Briggs";

auto arr1 = name;              // arr1's type is const char*

auto& arr2 = name;             // arr2's type is
                               // const char (&)[13]


void someFunc(int, double);    // someFunc is a function;
                               // type is void(int, double)

auto func1 = someFunc;         // func1's type is
                               // void (*)(int, double)

auto& func2 = someFunc;        // func2's type is
                               // void (&)(int, double)
```

As you can see, `auto` type deduction works like template type deduction. They're essentially two sides of the same coin.

Except for the one way they differ. We'll start with the observation that if you want to declare an `int` with an initial value of 27, C++98 gives you two syntactic choices:

```
int x1 = 27;
int x2(27);
```

C++11, through its support for uniform initialization, adds these:

```
int x3 = { 27 };
int x4{ 27 };
```

All in all, four syntaxes, but only one result: an `int` with value 27.

But as Item 5 explains, there are advantages to declaring variables using `auto` instead of fixed types, so it'd be nice to replace `int` with `auto` in the above variable declarations. Straightforward textual substitution yields this code:

```
auto x1 = 27;
auto x2(27);
auto x3 = { 27 };
auto x4{ 27 };
```

These declarations all compile, but they don't have the same meaning as the ones they replace. The first two statements do, indeed, declare a variable of type `int` with value 27. The second two, however, declare a variable of type `std::initializer_list<int>` containing a single element with value 27!

```
auto x1 = 27;             // type is int, value is 27

auto x2(27);              // ditto

auto x3 = { 27 };         // type is std::initializer_list<int>,
                          // value is { 27 }

auto x4{ 27 };            // ditto
```

This is due to a special type deduction rule for `auto`. When the initializer for an `auto`-declared variable is enclosed in braces, the deduced type is a `std::initializer_list`. If such a type can't be deduced (e.g., because the values in the braced initializer are of different types), the code will be rejected:

```
auto x5 = { 1, 2, 3.0 };  // error! can't deduce T for
                          // std::initializer_list<T>
```

As the comment indicates, type deduction will fail in this case, but it's important to recognize that there are actually two kinds of type deduction taking place. One kind stems from the use of `auto`: `x5`'s type has to be deduced. Because `x5`'s initializer is in braces, `x5` must be deduced to be a `std::initializer_list`. But `std::initializer_list` is a template. Instantiations are `std::initializer_list<T>` for some type `T`, and that means that `T`'s type must also be deduced. Such deduction falls under the purview of the second kind of type deduction occurring here: template type deduction. In this example, that deduction fails, because the values in the braced initializer don't have a single type.

The treatment of braced initializers is the only way in which `auto` type deduction and template type deduction differ. When an `auto`–declared variable is initialized with a braced initializer, the deduced type is an instantiation of `std::initializer_list`. But if the corresponding template is passed the same initializer, type deduction fails, and the code is rejected:

```
auto x = { 11, 23, 9 };    // x's type is
                           // std::initializer_list<int>

template<typename T>       // template with parameter
void f(T param);           // declaration equivalent to
                           // x's declaration

f({ 11, 23, 9 });          // error! can't deduce type for T
```

However, if you specify in the template that `param` is a `std::initializer_list<T>` for some unknown `T`, template type deduction will deduce what `T` is:

```
template<typename T>
void f(std::initializer_list<T> initList);

f({ 11, 23, 9 });          // T deduced as int, and initList's
                           // type is std::initializer_list<int>
```

So the only real difference between `auto` and template type deduction is that `auto` *assumes* that a braced initializer represents a `std::initializer_list`, but template type deduction doesn't.

You might wonder why `auto` type deduction has a special rule for braced initializers, but template type deduction does not. I wonder this myself. Alas, I have not been able to find a convincing explanation. But the rule is the rule, and this means you must remember that if you declare a variable using `auto` and you initialize it with a braced initializer, the deduced type will always be `std::initializer_list`. It's especially important to bear this in mind if you embrace the philosophy of uniform initialization—of enclosing initializing values in braces as a matter of course. A classic mistake in C++11 programming is accidentally declaring a `std::initializer_list` variable when you mean to declare something else. This pitfall is one of the reasons some developers put braces around their initializers only when they have to. (When you have to is discussed in Item 7.)

For C++11, this is the full story, but for C++14, the tale continues. C++14 permits `auto` to indicate that a function's return type should be deduced (see Item 3), and C++14 lambdas may use `auto` in parameter declarations. However, these uses of `auto` employ *template type deduction*, not `auto` type deduction. So a function with an `auto` return type that returns a braced initializer won't compile:

```
auto createInitList()
{
  return { 1, 2, 3 };          // error: can't deduce type
}                              // for { 1, 2, 3 }
```

The same is true when `auto` is used in a parameter type specification in a C++14 lambda:

```
std::vector<int> v;
…

auto resetV =
  [&v](const auto& newValue) { v = newValue; };     // C++14

…

resetV({ 1, 2, 3 });           // error! can't deduce type
                               // for { 1, 2, 3 }
```

Things to Remember

- `auto` type deduction is usually the same as template type deduction, but `auto` type deduction assumes that a braced initializer represents a `std::initializer_list`, and template type deduction doesn't.
- `auto` in a function return type or a lambda parameter implies template type deduction, not `auto` type deduction.

Item 3: Understand `decltype`.

`decltype` is an odd creature. Given a name or an expression, `decltype` tells you the name's or the expression's type. Typically, what it tells you is exactly what you'd predict. Occasionally however, it provides results that leave you scratching your head and turning to reference works or online Q&A sites for revelation.

We'll begin with the typical cases—the ones harboring no surprises. In contrast to what happens during type deduction for templates and `auto` (see Items 1 and 2), `decltype` typically parrots back the exact type of the name or expression you give it:

```
const int i = 0;           // decltype(i) is const int

bool f(const Widget& w);   // decltype(w) is const Widget&
                           // decltype(f) is bool(const Widget&)

struct Point {
  int x, y;                // decltype(Point::x) is int
};                         // decltype(Point::y) is int

Widget w;                  // decltype(w) is Widget

if (f(w)) …                // decltype(f(w)) is bool

template<typename T>       // simplified version of std::vector
class vector {
public:
  …
  T& operator[](std::size_t index);
  …
};

vector<int> v;             // decltype(v) is vector<int>
…
if (v[0] == 0) …           // decltype(v[0]) is int&
```

See? No surprises.

In C++11, perhaps the primary use for `decltype` is declaring function templates where the function's return type depends on its parameter types. For example, suppose we'd like to write a function that takes a container that supports indexing via square brackets (i.e., the use of "`[]`") plus an index, then authenticates the user before returning the result of the indexing operation. The return type of the function should be the same as the type returned by the indexing operation.

`operator[]` on a container of objects of type `T` typically returns a `T&`. This is the case for `std::deque`, for example, and it's almost always the case for `std::vector`. For `std::vector<bool>`, however, `operator[]` does not return a `bool&`. Instead, it returns a brand new object. The whys and hows of this situation are explored in

Item 6, but what's important here is that the type returned by a container's `operator[]` depends on the container.

`decltype` makes it easy to express that. Here's a first cut at the template we'd like to write, showing the use of `decltype` to compute the return type. The template needs a bit of refinement, but we'll defer that for now:

```
template<typename Container, typename Index>    // works, but
auto authAndAccess(Container& c, Index i)       // requires
  -> decltype(c[i])                             // refinement
{
  authenticateUser();
  return c[i];
}
```

The use of `auto` before the function name has nothing to do with type deduction. Rather, it indicates that C++11's *trailing return type* syntax is being used, i.e., that the function's return type will be declared following the parameter list (after the "`->`"). A trailing return type has the advantage that the function's parameters can be used in the specification of the return type. In `authAndAccess`, for example, we specify the return type using `c` and `i`. If we were to have the return type precede the function name in the conventional fashion, `c` and `i` would be unavailable, because they would not have been declared yet.

With this declaration, `authAndAccess` returns whatever type `operator[]` returns when applied to the passed-in container, exactly as we desire.

C++11 permits return types for single-statement lambdas to be deduced, and C++14 extends this to both all lambdas and all functions, including those with multiple statements. In the case of `authAndAccess`, that means that in C++14 we can omit the trailing return type, leaving just the leading `auto`. With that form of declaration, `auto` *does* mean that type deduction will take place. In particular, it means that compilers will deduce the function's return type from the function's implementation:

```
template<typename Container, typename Index>    // C++14;
auto authAndAccess(Container& c, Index i)       // not quite
{                                               // correct
  authenticateUser();
  return c[i];                  // return type deduced from c[i]
}
```

Item 2 explains that for functions with an `auto` return type specification, compilers employ template type deduction. In this case, that's problematic. As we've discussed, `operator[]` for most containers-of-`T` returns a `T&`, but Item 1 explains that during template type deduction, the reference-ness of an initializing expression is ignored. Consider what that means for this client code:

```
std::deque<int> d;
…
authAndAccess(d, 5) = 10;  // authenticate user, return d[5],
                           // then assign 10 to it;
                           // this won't compile!
```

Here, `d[5]` returns an `int&`, but `auto` return type deduction for `authAndAccess` will strip off the reference, thus yielding a return type of `int`. That `int`, being the return value of a function, is an rvalue, and the code above thus attempts to assign 10 to an rvalue `int`. That's forbidden in C++, so the code won't compile.

To get `authAndAccess` to work as we'd like, we need to use `decltype` type deduction for its return type, i.e., to specify that `authAndAccess` should return exactly the same type that the expression `c[i]` returns. The guardians of C++, anticipating the need to use `decltype` type deduction rules in some cases where types are inferred, make this possible in C++14 through the `decltype(auto)` specifier. What may initially seem contradictory (`decltype` *and* `auto`?) actually makes perfect sense: `auto` specifies that the type is to be deduced, and `decltype` says that `decltype` rules should be used during the deduction. We can thus write `authAndAccess` like this:

```
template<typename Container, typename Index>   // C++14; works,
decltype(auto)                                 // but still
authAndAccess(Container& c, Index i)           // requires
{                                              // refinement
  authenticateUser();
  return c[i];
}
```

Now `authAndAccess` will truly return whatever `c[i]` returns. In particular, for the common case where `c[i]` returns a `T&`, `authAndAccess` will also return a `T&`, and in the uncommon case where `c[i]` returns an object, `authAndAccess` will return an object, too.

The use of `decltype(auto)` is not limited to function return types. It can also be convenient for declaring variables when you want to apply the `decltype` type deduction rules to the initializing expression:

```
Widget w;

const Widget& cw = w;

auto myWidget1 = cw;             // auto type deduction:
```

```
                                    // myWidget1's type is Widget

decltype(auto) myWidget2 = cw;      // decltype type deduction:
                                    // myWidget2's type is
                                    //   const Widget&
```

But two things are bothering you, I know. One is the refinement to `authAndAccess` I mentioned, but have not yet described. Let's address that now.

Look again at the declaration for the C++14 version of `authAndAccess`:

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i);
```

The container is passed by lvalue-reference-to-non-`const`, because returning a reference to an element of the container permits clients to modify that container. But this means it's not possible to pass rvalue containers to this function. Rvalues can't bind to lvalue references (unless they're lvalue-references-to-`const`, which is not the case here).

Admittedly, passing an rvalue container to `authAndAccess` is an edge case. An rvalue container, being a temporary object, would typically be destroyed at the end of the statement containing the call to `authAndAccess`, and that means that a reference to an element in that container (which is typically what `authAndAccess` would return) would dangle at the end of the statement that created it. Still, it could make sense to pass a temporary object to `authAndAccess`. A client might simply want to make a copy of an element in the temporary container, for example:

```
std::deque<std::string> makeStringDeque();   // factory function

// make copy of 5th element of deque returned
// from makeStringDeque
auto s = authAndAccess(makeStringDeque(), 5);
```

Supporting such use means we need to revise the declaration for `authAndAccess` to accept both lvalues and rvalues. Overloading would work (one overload would declare an lvalue reference parameter, the other an rvalue reference parameter), but then we'd have two functions to maintain. A way to avoid that is to have `authAndAccess` employ a reference parameter that can bind to lvalues *and* rvalues, and Item 24 explains that that's exactly what universal references do. `authAndAccess` can therefore be declared like this:

```
template<typename Container, typename Index>     // c is now a
decltype(auto) authAndAccess(Container&& c,      // universal
                             Index i);           // reference
```

In this template, we don't know what type of container we're operating on, and that means we're equally ignorant of the type of index objects it uses. Employing pass-by-value for objects of an unknown type generally risks the performance hit of unnecessary copying, the behavioral problems of object slicing (see Item 41), and the sting of our coworkers' derision, but in the case of container indices, following the example of the Standard Library for index values (e.g., in `operator[]` for `std::string`, `std::vector`, and `std::deque`) seems reasonable, so we'll stick with pass-by-value for them.

However, we need to update the template's implementation to bring it into accord with Item 25's admonition to apply `std::forward` to universal references:

```
template<typename Container, typename Index>       // final
decltype(auto)                                     // C++14
authAndAccess(Container&& c, Index i)              // version
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

This should do everything we want, but it requires a C++14 compiler. If you don't have one, you'll need to use the C++11 version of the template. It's the same as its C++14 counterpart, except that you have to specify the return type yourself:

```
template<typename Container, typename Index>       // final
auto                                               // C++11
authAndAccess(Container&& c, Index i)              // version
-> decltype(std::forward<Container>(c)[i])
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

The other issue that's likely to be nagging at you is my remark at the beginning of this Item that decltype *almost* always produces the type you expect, that it *rarely* surprises. Truth be told, you're unlikely to encounter these exceptions to the rule unless you're a heavy-duty library implementer.

To *fully* understand `decltype`'s behavior, you'll have to familiarize yourself with a few special cases. Most of these are too obscure to warrant discussion in a book like this, but looking at one lends insight into `decltype` as well as its use.

Applying `decltype` to a name yields the declared type for that name. Names are lvalue expressions, but that doesn't affect `decltype`'s behavior. For lvalue expressions more complicated than names, however, `decltype` ensures that the type reported is always an lvalue reference. That is, if an lvalue expression other than a name has type `T`, `decltype` reports that type as `T&`. This seldom has any impact, because the type of most lvalue expressions inherently includes an lvalue reference qualifier. Functions returning lvalues, for example, always return lvalue references.

There is an implication of this behavior that is worth being aware of, however. In

```
int x = 0;
```

`x` is the name of a variable, so `decltype(x)` is `int`. But wrapping the name `x` in parentheses—"`(x)`"—yields an expression more complicated than a name. Being a name, `x` is an lvalue, and C++ defines the expression `(x)` to be an lvalue, too. `decltype((x))` is therefore `int&`. Putting parentheses around a name can change the type that `decltype` reports for it!

In C++11, this is little more than a curiosity, but in conjunction with C++14's support for `decltype(auto)`, it means that a seemingly trivial change in the way you write a `return` statement can affect the deduced type for a function:

```
decltype(auto) f1()
{
  int x = 0;
  …
  return x;         // decltype(x) is int, so f1 returns int
}

decltype(auto) f2()
{
  int x = 0;
  …
  return (x);       // decltype((x)) is int&, so f2 returns int&
}
```

Note that not only does `f2` have a different return type from `f1`, it's also returning a reference to a local variable! That's the kind of code that puts you on the express train to undefined behavior—a train you certainly don't want to be on.

The primary lesson is to pay very close attention when using `decltype(auto)`. Seemingly insignificant details in the expression whose type is being deduced can affect the type that `decltype(auto)` reports. To ensure that the type being deduced is the type you expect, use the techniques described in Item 4.

At the same time, don't lose sight of the bigger picture. Sure, `decltype` (both alone and in conjunction with `auto`) may occasionally yield type-deduction surprises, but that's not the normal situation. Normally, `decltype` produces the type you expect.

This is especially true when `decltype` is applied to names, because in that case, `decltype` does just what it sounds like: it reports that name's declared type.

Things to Remember

- `decltype` almost always yields the type of a variable or expression without any modifications.
- For lvalue expressions of type `T` other than names, `decltype` always reports a type of `T&`.
- C++14 supports `decltype(auto)`, which, like `auto`, deduces a type from its initializer, but it performs the type deduction using the `decltype` rules.

Item 4: Know how to view deduced types.

The choice of tools for viewing the results of type deduction is dependent on the phase of the software development process where you want the information. We'll explore three possibilities: getting type deduction information as you edit your code, getting it during compilation, and getting it at runtime.

IDE Editors

Code editors in IDEs often show the types of program entities (e.g., variables, parameters, functions, etc.) when you do something like hover your cursor over the entity. For example, given this code,

```
const int theAnswer = 42;

auto x = theAnswer;
auto y = &theAnswer;
```

an IDE editor would likely show that `x`'s deduced type was `int` and `y`'s was `const int*`.

For this to work, your code must be in a more or less compilable state, because what makes it possible for the IDE to offer this kind of information is a C++ compiler (or at least the front end of one) running inside the IDE. If that compiler can't make enough sense of your code to parse it and perform type deduction, it can't show you what types it deduced.

For simple types like `int`, information from IDEs is generally fine. As we'll see soon, however, when more complicated types are involved, the information displayed by IDEs may not be particularly helpful.

Compiler Diagnostics

An effective way to get a compiler to show a type it has deduced is to use that type in a way that leads to compilation problems. The error message reporting the problem is virtually sure to mention the type that's causing it.

Suppose, for example, we'd like to see the types that were deduced for `x` and `y` in the previous example. We first declare a class template that we *don't define*. Something like this does nicely:

```
template<typename T>        // declaration only for TD;
class TD;                   // TD == "Type Displayer"
```

Attempts to instantiate this template will elicit an error message, because there's no template definition to instantiate. To see the types for `x` and `y`, just try to instantiate `TD` with their types:

```
TD<decltype(x)> xType;      // elicit errors containing
TD<decltype(y)> yType;      // x's and y's types
```

I use variable names of the form *`variableName`*`Type`, because they tend to yield error messages that help me find the information I'm looking for. For the code above, one of my compilers issues diagnostics reading, in part, as follows (I've highlighted the type information we're after):

```
error: aggregate 'TD<int> xType' has incomplete type and
    cannot be defined
error: aggregate 'TD<const int *> yType' has incomplete type
    and cannot be defined
```

A different compiler provides the same information, but in a different form:

```
error: 'xType' uses undefined class 'TD<int>'
error: 'yType' uses undefined class 'TD<const int *>'
```

Formatting differences aside, all the compilers I've tested produce error messages with useful type information when this technique is employed.

Runtime Output

The `printf` approach to displaying type information (not that I'm recommending you use `printf`) can't be employed until runtime, but it offers full control over the formatting of the output. The challenge is to create a textual representation of the type you care about that is suitable for display. "No sweat," you're thinking, "it's `typeid` and `std::type_info::name` to the rescue." In our continuing quest to see the types deduced for `x` and `y`, you may figure we can write this:

```
std::cout << typeid(x).name() << '\n';    // display types for
std::cout << typeid(y).name() << '\n';    // x and y
```

This approach relies on the fact that invoking typeid on an object such as x or y yields a std::type_info object, and std::type_info has a member function, name, that produces a C-style string (i.e., a const char*) representation of the name of the type.

Calls to std::type_info::name are not guaranteed to return anything sensible, but implementations try to be helpful. The level of helpfulness varies. The GNU and Clang compilers report that the type of x is "i", and the type of y is "PKi", for example. These results make sense once you learn that, in output from these compilers, "i" means "int" and "PK" means "pointer to ~~konst~~ const." (Both compilers support a tool, c++filt, that decodes such "mangled" types.) Microsoft's compiler produces less cryptic output: "int" for x and "int const *" for y.

Because these results are correct for the types of x and y, you might be tempted to view the type-reporting problem as solved, but let's not be hasty. Consider a more complex example:

```
template<typename T>                    // template function to
void f(const T& param);                 // be called

std::vector<Widget> createVec();        // factory function

const auto vw = createVec();            // init vw w/factory return

if (!vw.empty()) {
  f(&vw[0]);                            // call f
  …
}
```

This code, which involves a user-defined type (Widget), an STL container (std::vector), and an auto variable (vw), is more representative of the situations where you might want some visibility into the types your compilers are deducing. For example, it'd be nice to know what types are inferred for the template type parameter T and the function parameter param in f.

Loosing typeid on the problem is straightforward. Just add some code to f to display the types you'd like to see:

```
template<typename T>
void f(const T& param)
{
  using std::cout;
```

```
  cout << "T =     " << typeid(T).name() << '\n';     // show T

  cout << "param = " << typeid(param).name() << '\n'; // show
  …                                                   // param's
}                                                     // type
```

Executables produced by the GNU and Clang compilers produce this output:

```
T =     PK6Widget
param = PK6Widget
```

We already know that for these compilers, `PK` means "pointer to `const`," so the only mystery is the number `6`. That's simply the number of characters in the class name that follows (`Widget`). So these compilers tell us that both `T` and `param` are of type `const Widget*`.

Microsoft's compiler concurs:

```
T =     class Widget const *
param = class Widget const *
```

Three independent compilers producing the same information suggests that the information is accurate. But look more closely. In the template `f`, `param`'s declared type is `const T&`. That being the case, doesn't it seem odd that `T` and `param` have the same type? If `T` were `int`, for example, `param`'s type should be `const int&`—not the same type at all.

Sadly, the results of `std::type_info::name` are not reliable. In this case, for example, the type that all three compilers report for `param` are incorrect. Furthermore, they're essentially *required* to be incorrect, because the specification for `std::type_info::name` mandates that the type be treated as if it had been passed to a template function as a by-value parameter. As Item 1 explains, that means that if the type is a reference, its reference-ness is ignored, and if the type after reference removal is `const` (or `volatile`), its constness (or volatileness) is also ignored. That's why `param`'s type—which is `const Widget * const &`—is reported as `const Widget*`. First the type's reference-ness is removed, and then the constness of the resulting pointer is eliminated.

Equally sadly, the type information displayed by IDE editors is also not reliable—or at least not reliably useful. For this same example, one IDE editor I know reports `T`'s type as (I am not making this up):

```
const
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget> >::_Alloc>::value_type>::value_type *
```

The same IDE editor shows `param`'s type as:

```
const std::_Simple_types<...>::value_type *const &
```

That's less intimidating than the type for `T`, but the "`...`" in the middle is confusing until you realize that it's the IDE editor's way of saying "I'm omitting all that stuff that's part of `T`'s type." With any luck, your development environment does a better job on code like this.

If you're more inclined to rely on libraries than luck, you'll be pleased to know that where `std::type_info::name` and IDEs may fail, the Boost TypeIndex library (often written as *Boost.TypeIndex*) is designed to succeed. The library isn't part of Standard C++, but neither are IDEs or templates like `TD`. Furthermore, the fact that Boost libraries (available at *boost.com*) are cross-platform, open source, and available under a license designed to be palatable to even the most paranoid corporate legal team means that code using Boost libraries is nearly as portable as code relying on the Standard Library.

Here's how our function `f` can produce accurate type information using Boost.TypeIndex:

```
#include <boost/type_index.hpp>

template<typename T>
void f(const T& param)
{
  using std::cout;
  using boost::typeindex::type_id_with_cvr;

  // show T
  cout << "T =     "
       << type_id_with_cvr<T>().pretty_name()
       << '\n';

  // show param's type
  cout << "param = "
       << type_id_with_cvr<decltype(param)>().pretty_name()
       << '\n';
  …
}
```

The way this works is that the function template `boost::typeindex::type_id_with_cvr` takes a type argument (the type about which we want information) and *doesn't* remove `const`, `volatile`, or reference qualifiers (hence the "`with_cvr`" in the template name). The result is a `boost::typeindex::type_index` object, whose `pretty_name` member function produces a `std::string` containing a human-friendly representation of the type.

With this implementation for `f`, consider again the call that yields incorrect type information for `param` when `typeid` is used:

```
std::vector<Widget> createVec();      // factory function

const auto vw = createVec();          // init vw w/factory return

if (!vw.empty()) {
  f(&vw[0]);                          // call f
  …
}
```

Under compilers from GNU and Clang, Boost.TypeIndex produces this (accurate) output:

```
T =     Widget const*
param = Widget const* const&
```

Results under Microsoft's compiler are essentially the same:

```
T =     class Widget const *
param = class Widget const * const &
```

Such near-uniformity is nice, but it's important to remember that IDE editors, compiler error messages, and libraries like Boost.TypeIndex are merely tools you can use to help you figure out what types your compilers are deducing. All can be helpful, but at the end of the day, there's no substitute for understanding the type deduction information in Items 1–3.

Things to Remember

- Deduced types can often be seen using IDE editors, compiler error messages, and the Boost TypeIndex library.
- The results of some tools may be neither helpful nor accurate, so an understanding of C++'s type deduction rules remains essential.

CHAPTER 2

auto

In concept, `auto` is as simple as simple can be, but it's more subtle than it looks. Using it saves typing, sure, but it also prevents correctness and performance issues that can bedevil manual type declarations. Furthermore, some of `auto`'s type deduction results, while dutifully conforming to the prescribed algorithm, are, from the perspective of a programmer, just wrong. When that's the case, it's important to know how to guide `auto` to the right answer, because falling back on manual type declarations is an alternative that's often best avoided.

This brief chapter covers all of `auto`'s ins and outs.

Item 5: Prefer `auto` to explicit type declarations.

Ah, the simple joy of

```
int x;
```

Wait. Damn. I forgot to initialize `x`, so its value is indeterminate. Maybe. It might actually be initialized to zero. Depends on the context. Sigh.

Never mind. Let's move on to the simple joy of declaring a local variable to be initialized by dereferencing an iterator:

```
template<typename It>    // algorithm to dwim ("do what I mean")
void dwim(It b, It e)    // for all elements in range from
{                        // b to e
  while (b != e) {
    typename std::iterator_traits<It>::value_type
      currValue = *b;
    …
```

```
    }
  }
```

Ugh. "`typename std::iterator_traits<It>::value_type`" to express the type of the value pointed to by an iterator? Really? I must have blocked out the memory of how much fun that is. Damn. Wait—didn't I already say that?

Okay, simple joy (take three): the delight of declaring a local variable whose type is that of a closure. Oh, right. The type of a closure is known only to the compiler, hence can't be written out. Sigh. Damn.

Damn, damn, damn! Programming in C++ is not the joyous experience it should be!

Well, it didn't used to be. But as of C++11, all these issues go away, courtesy of `auto`. `auto` variables have their type deduced from their initializer, so they must be initialized. That means you can wave goodbye to a host of uninitialized variable problems as you speed by on the modern C++ superhighway:

```
int x1;                       // potentially uninitialized

auto x2;                      // error! initializer required

auto x3 = 0;                  // fine, x's value is well-defined
```

Said highway lacks the potholes associated with declaring a local variable whose value is that of a dereferenced iterator:

```
template<typename It>         // as before
void dwim(It b, It e)
{
  while (b != e) {
    auto currValue = *b;
    …
  }
}
```

And because `auto` uses type deduction (see Item 2), it can represent types known only to compilers:

```
auto derefUPLess =                            // comparison func.
  [](const std::unique_ptr<Widget>& p1,       // for Widgets
     const std::unique_ptr<Widget>& p2)       // pointed to by
  { return *p1 < *p2; };                      // std::unique_ptrs
```

Very cool. In C++14, the temperature drops further, because parameters to lambda expressions may involve `auto`:

```
auto derefLess =                              // C++14 comparison
  [](const auto& p1,                          // function for
```

```
      const auto& p2)                   // values pointed
    { return *p1 < *p2; };              // to by anything
                                        // pointer-like
```

Coolness notwithstanding, perhaps you're thinking we don't really need `auto` to declare a variable that holds a closure, because we can use a `std::function` object. It's true, we can, but possibly that's not what you were thinking. And maybe now you're thinking "What's a `std::function` object?" So let's clear that up.

`std::function` is a template in the C++11 Standard Library that generalizes the idea of a function pointer. Whereas function pointers can point only to functions, however, `std::function` objects can refer to any callable object, i.e., to anything that can be invoked like a function. Just as you must specify the type of function to point to when you create a function pointer (i.e., the signature of the functions you want to point to), you must specify the type of function to refer to when you create a `std::function` object. You do that through `std::function`'s template parameter. For example, to declare a `std::function` object named `func` that could refer to any callable object acting as if it had this signature,

```
bool(const std::unique_ptr<Widget>&,  // C++11 signature for
     const std::unique_ptr<Widget>&)  // std::unique_ptr<Widget>
                                      // comparison function
```

you'd write this:

```
std::function<bool(const std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)> func;
```

Because lambda expressions yield callable objects, closures can be stored in `std::function` objects. That means we could declare the C++11 version of `derefUPLess` without using `auto` as follows:

```
std::function<bool(const std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)>
  derefUPLess = [](const std::unique_ptr<Widget>& p1,
                   const std::unique_ptr<Widget>& p2)
                 { return *p1 < *p2; };
```

It's important to recognize that even setting aside the syntactic verbosity and need to repeat the parameter types, using `std::function` is not the same as using `auto`. An `auto`-declared variable holding a closure has the same type as the closure, and as such it uses only as much memory as the closure requires. The type of a `std::function`-declared variable holding a closure is an instantiation of the `std::function` template, and that has a fixed size for any given signature. This size may not be adequate for the closure it's asked to store, and when that's the case, the `std::function` constructor will allocate heap memory to store the closure. The result is that the std::function object typically uses more memory than the auto-declared object. And, thanks to implementation details that restrict inlining and yield indirect function calls, invoking a closure via a std::function object is almost certain to be slower than calling it via an auto-declared object. In other words, the std::function approach is generally bigger and slower than the auto approach, and it may yield out-of-memory exceptions, too. Plus, as you can see in the examples above, writing "auto" is a whole lot less work than writing the type of the std::function instantiation. In the competition between auto and std::function for holding a closure, it's pretty much game, set, and match for auto. (A similar argument can be made for auto over std::function for holding the result of calls to std::bind, but in Item 34, I do my best to convince you to use lambdas instead of std::bind, anyway.)

The advantages of auto extend beyond the avoidance of uninitialized variables, verbose variable declarations, and the ability to directly hold closures. One is the ability to avoid what I call problems related to "type shortcuts." Here's something you've probably seen—possibly even written:

```
std::vector<int> v;
…
unsigned sz = v.size();
```

The official return type of v.size() is std::vector<int>::size_type, but few developers are aware of that. std::vector<int>::size_type is specified to be an unsigned integral type, so a lot of programmers figure that unsigned is good enough and write code such as the above. This can have some interesting consequences. On 32-bit Windows, for example, both unsigned and std::vector<int>::size_type are the same size, but on 64-bit Windows, unsigned is 32 bits, while std::vector<int>::size_type is 64 bits. This means that code that works under 32-bit Windows may behave incorrectly under 64-bit Windows, and when porting your application from 32 to 64 bits, who wants to spend time on issues like that?

Using auto ensures that you don't have to:

```
auto sz = v.size();  // sz's type is std::vector<int>::size_type
```

Still unsure about the wisdom of using auto? Then consider this code:

```
std::unordered_map<std::string, int> m;
…

for (const std::pair<std::string, int>& p : m)
{
  …                     // do something with p
}
```

This looks perfectly reasonable, but there's a problem. Do you see it?

Recognizing what's amiss requires remembering that the key part of a `std::unordered_map` is `const`, so the type of `std::pair` in the hash table (which is what a `std::unordered_map` is) isn't `std::pair<std::string, int>`, it's `std::pair<const std::string, int>`. But that's not the type declared for the variable `p` in the loop above. As a result, compilers will strive to find a way to convert `std::pair<const std::string, int>` objects (i.e., what's in the hash table) to `std::pair<std::string, int>` objects (the declared type for `p`). They'll succeed by creating a temporary object of the type that `p` wants to bind to by copying each object in `m`, then binding the reference `p` to that temporary object. At the end of each loop iteration, the temporary object will be destroyed. If you wrote this loop, you'd likely be surprised by this behavior, because you'd almost certainly intend to simply bind the reference `p` to each element in `m`.

Such unintentional type mismatches can be `auto`ed away:

```
for (const auto& p : m)
{
  …                                 // as before
}
```

This is not only more efficient, it's also easier to type. Furthermore, this code has the very attractive characteristic that if you take `p`'s address, you're sure to get a pointer to an element within `m`. In the code not using `auto`, you'd get a pointer to a temporary object—an object that would be destroyed at the end of the loop iteration.

The last two examples—writing `unsigned` when you should have written `std::vector<int>::size_type` and writing `std::pair<std::string, int>` when you should have written `std::pair<const std::string, int>`—demonstrate how explicitly specifying types can lead to implicit conversions that you neither want nor expect. If you use `auto` as the type of the target variable, you need not worry about mismatches between the type of variable you're declaring and the type of the expression used to initialize it.

There are thus several reasons to prefer `auto` over explicit type declarations. Yet `auto` isn't perfect. The type for each `auto` variable is deduced from its initializing expression, and some initializing expressions have types that are neither anticipated nor desired. The conditions under which such cases arise, and what you can do about them, are discussed in Items 2 and 6, so I won't address them here. Instead, I'll turn my attention to a different concern you may have about using `auto` in place of traditional type declarations: the readability of the resulting source code.

First, take a deep breath and relax. `auto` is an option, not a mandate. If, in your professional judgment, your code will be clearer or more maintainable or in some other way better by using explicit type declarations, you're free to continue using them. But bear in mind that C++ breaks no new ground in adopting what is generally known in the programming languages world as *type inference*. Other statically typed procedural languages (e.g., C#, D, Scala, Visual Basic) have a more or less equivalent feature, to say nothing of a variety of statically typed functional languages (e.g., ML, Haskell, OCaml, F#, etc.). In part, this is due to the success of dynamically typed languages such as Perl, Python, and Ruby, where variables are rarely explicitly typed. The software development community has extensive experience with type inference, and it has demonstrated that there is nothing contradictory about such technology and the creation and maintenance of large, industrial-strength code bases.

Some developers are disturbed by the fact that using `auto` eliminates the ability to determine an object's type by a quick glance at the source code. However, IDEs' ability to show object types often mitigates this problem (even taking into account the IDE type-display issues mentioned in Item 4), and, in many cases, a somewhat abstract view of an object's type is just as useful as the exact type. It often suffices, for example, to know that an object is a container or a counter or a smart pointer, without knowing exactly what kind of container, counter, or smart pointer it is. Assuming well-chosen variable names, such abstract type information should almost always be at hand.

The fact of the matter is that writing types explicitly often does little more than introduce opportunities for subtle errors, either in correctness or efficiency or both. Furthermore, `auto` types automatically change if the type of their initializing expression changes, and that means that some refactorings are facilitated by the use of `auto`. For example, if a function is declared to return an `int`, but you later decide that a `long` would be better, the calling code automatically updates itself the next time you compile if the results of calling the function are stored in `auto` variables. If the results are stored in variables explicitly declared to be `int`, you'll need to find all the call sites so that you can revise them.

Things to Remember

- `auto` variables must be initialized, are generally immune to type mismatches that can lead to portability or efficiency problems, can ease the process of refactoring, and typically require less typing than variables with explicitly specified types.
- `auto`-typed variables are subject to the pitfalls described in Items 2 and 6.

Item 6: Use the explicitly typed initializer idiom when auto deduces undesired types.

Item 5 explains that using `auto` to declare variables offers a number of technical advantages over explicitly specifying types, but sometimes `auto`'s type deduction zigs when you want it to zag. For example, suppose I have a function that takes a `Widget` and returns a `std::vector<bool>`, where each `bool` indicates whether the `Widget` offers a particular feature:

```
std::vector<bool> features(const Widget& w);
```

Further suppose that bit 5 indicates whether the `Widget` has high priority. We can thus write code like this:

```
Widget w;
…

bool highPriority = features(w)[5];  // is w high priority?
…

processWidget(w, highPriority);      // process w in accord
                                     // with its priority
```

There's nothing wrong with this code. It'll work fine. But if we make the seemingly innocuous change of replacing the explicit type for `highPriority` with `auto`,

```
auto highPriority = features(w)[5];  // is w high priority?
```

the situation changes. All the code will continue to compile, but its behavior is no longer predictable:

```
processWidget(w, highPriority);      // undefined behavior!
```

As the comment indicates, the call to `processWidget` now has undefined behavior. But why? The answer is likely to be surprising. In the code using `auto`, the type of `highPriority` is no longer `bool`. Though `std::vector<bool>` conceptually holds `bool`s, `operator[]` for `std::vector<bool>` doesn't return a reference to an element of the container (which is what `std::vector::operator[]` returns for every type *except* `bool`). Instead, it returns an object of type `std::vector<bool>::reference` (a class nested inside `std::vector<bool>`).

`std::vector<bool>::reference` exists because `std::vector<bool>` is specified to represent its `bool`s in packed form, one bit per `bool`. That creates a problem for `std::vector<bool>`'s `operator[]`, because `operator[]` for `std::vector<T>` is supposed to return a `T&`, but C++ forbids references to bits. Not being able to return a `bool&`, `operator[]` for `std::vector<bool>` returns an object that *acts like* a `bool&`. For this act to succeed, `std::vector<bool>::reference` objects must be usable in essentially all contexts where `bool&`s can be. Among the features in `std::vector<bool>::reference` that make this work is an implicit conversion to `bool`. (Not to `bool&`, to *`bool`*. To explain the full set of techniques used by `std::vector<bool>::reference` to emulate the behavior of a `bool&` would take us too far afield, so I'll simply remark that this implicit conversion is only one stone in a larger mosaic.)

With this information in mind, look again at this part of the original code:

```
bool highPriority = features(w)[5];   // declare highPriority's
                                      // type explicitly
```

Here, `features` returns a `std::vector<bool>` object, on which `operator[]` is invoked. `operator[]` returns a `std::vector<bool>::reference` object, which is then implicitly converted to the `bool` that is needed to initialize `highPriority`. `highPriority` thus ends up with the value of bit 5 in the `std::vector<bool>` returned by `features`, just like it's supposed to.

Contrast that with what happens in the `auto`-ized declaration for `highPriority`:

```
auto highPriority = features(w)[5];   // deduce highPriority's
                                      // type
```

Again, `features` returns a `std::vector<bool>` object, and, again, `operator[]` is invoked on it. `operator[]` continues to return a `std::vector<bool>::reference` object, but now there's a change, because `auto` deduces that as the type of `highPriority`. `highPriority` doesn't have the value of bit 5 of the `std::vector<bool>` returned by `features` at all.

The value it does have depends on how `std::vector<bool>::reference` is implemented. One implementation is for such objects to contain a pointer to the machine word holding the referenced bit, plus the offset into that word for that bit. Consider what that means for the initialization of `highPriority`, assuming that such a `std::vector<bool>::reference` implementation is in place.

The call to `features` returns a temporary `std::vector<bool>` object. This object has no name, but for purposes of this discussion, I'll call it *`temp`*. `operator[]` is invoked on *`temp`*, and the `std::vector<bool>::reference` it returns contains a pointer to a word in the data structure holding the bits that are managed by *`temp`*, plus the offset into that word corresponding to bit 5. `highPriority` is a copy of this `std::vector<bool>::reference` object, so `highPriority`, too, contains a pointer to a word in *`temp`*, plus the offset corresponding to bit 5. At the end of the statement, *temp* is destroyed, because it's a temporary object. Therefore, `highPriority` contains a dangling pointer, and that's the cause of the undefined behavior in the call to `processWidget`:

```
processWidget(w, highPriority);    // undefined behavior!
                                   // highPriority contains
                                   // dangling pointer!
```

`std::vector<bool>::reference` is an example of a *proxy class*: a class that exists for the purpose of emulating and augmenting the behavior of some other type. Proxy classes are employed for a variety of purposes. `std::vector<bool>::reference` exists to offer the illusion that `operator[]` for `std::vector<bool>` returns a reference to a bit, for example, and the Standard Library's smart pointer types (see Chapter 4) are proxy classes that graft resource management onto raw pointers. The utility of proxy classes is well-established. In fact, the design pattern "Proxy" is one of the most longstanding members of the software design patterns Pantheon.

Some proxy classes are designed to be apparent to clients. That's the case for `std::shared_ptr` and `std::unique_ptr`, for example. Other proxy classes are designed to act more or less invisibly. `std::vector<bool>::reference` is an example of such "invisible" proxies, as is its `std::bitset` compatriot, `std::bitset::reference`.

Also in that camp are some classes in C++ libraries employing a technique known as *expression templates*. Such libraries were originally developed to improve the efficiency of numeric code. Given a class `Matrix` and `Matrix` objects `m1`, `m2`, `m3`, and `m4`, for example, the expression

```
Matrix sum = m1 + m2 + m3 + m4;
```

can be computed much more efficiently if `operator+` for `Matrix` objects returns a proxy for the result instead of the result itself. That is, `operator+` for two `Matrix` objects would return an object of a proxy class such as `Sum<Matrix, Matrix>` instead of a `Matrix` object. As was the case with `std::vector<bool>::reference` and `bool`, there'd be an implicit conversion from the proxy class to `Matrix`, which would permit the initialization of `sum` from the proxy object produced by the expression on the right side of the "=". (The type of that object would traditionally encode the entire initialization expression, i.e., be something like `Sum<Sum<Sum<Matrix, Matrix>, Matrix>, Matrix>`. That's definitely a type from which clients should be shielded.)

As a general rule, "invisible" proxy classes don't play well with `auto`. Objects of such classes are often not designed to live longer than a single statement, so creating variables of those types tends to violate fundamental library design assumptions. That's the case with `std::vector<bool>::reference`, and we've seen that violating that assumption can lead to undefined behavior.

You therefore want to avoid code of this form:

```
auto someVar = expression of "invisible" proxy class type;
```

But how can you recognize when proxy objects are in use? The software employing them is unlikely to advertise their existence. They're supposed to be *invisible*, at least conceptually! And once you've found them, do you really have to abandon `auto` and the many advantages Item 5 demonstrates for it?

Let's take the how-do-you-find-them question first. Although "invisible" proxy classes are designed to fly beneath programmer radar in day-to-day use, libraries using them often document that they do so. The more you've familiarized yourself with the basic design decisions of the libraries you use, the less likely you are to be blindsided by proxy usage within those libraries.

Where documentation comes up short, header files fill the gap. It's rarely possible for source code to fully cloak proxy objects. They're typically returned from functions that clients are expected to call, so function signatures usually reflect their existence. Here's the spec for `std::vector<bool>::operator[]`, for example:

```
namespace std {                                  // from C++ Standards

  template <class Allocator>
  class vector<bool, Allocator> {
  public:
    …
    class reference { … };

    reference operator[](size_type n);
    …
  };
}
```

Assuming you know that `operator[]` for `std::vector<T>` normally returns a `T&`, the unconventional return type for `operator[]` in this case is a tip-off that a proxy class is in use. Paying careful attention to the interfaces you're using can often reveal the existence of proxy classes.

In practice, many developers discover the use of proxy classes only when they try to track down mystifying compilation problems or debug incorrect unit test results. Regardless of how you find them, once `auto` has been determined to be deducing the type of a proxy class instead of the type being proxied, the solution need not involve abandoning `auto`. `auto` itself isn't the problem. The problem is that `auto` isn't deducing the type you want it to deduce. The solution is to force a different type deduction. The way you do that is what I call *the explicitly typed initializer idiom*.

The explicitly typed initializer idiom involves declaring a variable with `auto`, but casting the initialization expression to the type you want `auto` to deduce. Here's how it can be used to force `highPriority` to be a `bool`, for example:

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

Here, `features(w)[5]` continues to return a `std::vector<bool>::reference` object, just as it always has, but the cast changes the type of the expression to `bool`, which `auto` then deduces as the type for `highPriority`. At runtime, the `std::vector<bool>::reference` object returned from `std::vector<bool>::operator[]` executes the conversion to `bool` that it supports, and as part of that conversion, the still-valid pointer to the `std::vector<bool>` returned from `features` is dereferenced. That avoids the undefined behavior we ran into earlier. The index 5 is then applied to the bits pointed to by the pointer, and the `bool` value that emerges is used to initialize `highPriority`.

For the `Matrix` example, the explicitly typed initializer idiom would look like this:

```
auto sum = static_cast<Matrix>(m1 + m2 + m3 + m4);
```

Applications of the idiom aren't limited to initializers yielding proxy class types. It can also be useful to emphasize that you are deliberately creating a variable of a type that is different from that generated by the initializing expression. For example, suppose you have a function to calculate some tolerance value:

```
double calcEpsilon();              // return tolerance value
```

`calcEpsilon` clearly returns a `double`, but suppose you know that for your application, the precision of a `float` is adequate, and you care about the difference in size between `float`s and `double`s. You could declare a `float` variable to store the result of `calcEpsilon`,

```
float ep = calcEpsilon();          // impliclitly convert
                                   // double → float
```

but this hardly announces "I'm deliberately reducing the precision of the value returned by the function." A declaration using the explicitly typed initializer idiom, however, does:

```
auto ep = static_cast<float>(calcEpsilon());
```

Similar reasoning applies if you have a floating-point expression that you are deliberately storing as an integral value. Suppose you need to calculate the index of an element in a container with random access iterators (e.g., a `std::vector`, `std::deque`, or `std::array`), and you're given a `double` between `0.0` and `1.0` indicating how far from the beginning of the container the desired element is located. (`0.5` would indicate the middle of the container.) Further suppose that you're confident that the resulting index will fit in an `int`. If the container is `c` and the double is `d`, you could calculate the index this way,

```
int index = d * c.size();
```

but this obscures the fact that you're intentionally converting the `double` on the right to an `int`. The explicitly typed initializer idiom makes things transparent:

```
auto index = static_cast<int>(d * c.size());
```

Things to Remember

- "Invisible" proxy types can cause `auto` to deduce the "wrong" type for an initializing expression.
- The explicitly typed initializer idiom forces `auto` to deduce the type you want it to have.

CHAPTER 3
Moving to Modern C++

When it comes to big-name features, C++11 and C++14 have a lot to boast of. `auto`, smart pointers, move semantics, lambdas, concurrency—each is so important, I devote a chapter to it. It's essential to master those features, but becoming an effective modern C++ programmer requires a series of smaller steps, too. Each step answers specific questions that arise during the journey from C++98 to modern C++. When should you use braces instead of parentheses for object creation? Why are alias declarations better than `typedef`s? How does `constexpr` differ from `const`? What's the relationship between `const` member functions and thread safety? The list goes on and on. And one by one, this chapter provides the answers.

Item 7: Distinguish between `()` and `{}` when creating objects.

Depending on your perspective, syntax choices for object initialization in C++11 embody either an embarrassment of riches or a confusing mess. As a general rule, initialization values may be specified with parentheses, an equals sign, or braces:

```
int x(0);               // initializer is in parentheses

int y = 0;              // initializer follows "="

int z{ 0 };             // initializer is in braces
```

In many cases, it's also possible to use an equals sign and braces together:

```
int z = { 0 };          // initializer uses "=" and braces
```

For the remainder of this Item, I'll generally ignore the equals-sign-plus-braces syntax, because C++ usually treats it the same as the braces-only version.

The “confusing mess” lobby points out that the use of an equals sign for initialization often misleads C++ newbies into thinking that an assignment is taking place, even though it’s not. For built-in types like `int`, the difference is academic, but for user-defined types, it’s important to distinguish initialization from assignment, because different function calls are involved:

```
Widget w1;             // call default constructor

Widget w2 = w1;        // not an assignment; calls copy ctor

w1 = w2;               // an assignment; calls copy operator=
```

Even with several initialization syntaxes, there were some situations where C++98 had no way to express a desired initialization. For example, it wasn’t possible to directly indicate that an STL container should be created holding a particular set of values (e.g., 1, 3, and 5).

To address the confusion of multiple initialization syntaxes, as well as the fact that they don’t cover all initialization scenarios, C++11 introduces *uniform initialization*: a single initialization syntax that can, at least in concept, be used anywhere and express everything. It’s based on braces, and for that reason I prefer the term *braced initialization*. “Uniform initialization” is an idea. “Braced initialization” is a syntactic construct.

Braced initialization lets you express the formerly inexpressible. Using braces, specifying the initial contents of a container is easy:

```
std::vector<int> v{ 1, 3, 5 }; // v's initial content is 1, 3, 5
```

Braces can also be used to specify default initialization values for non-static data members. This capability—new to C++11—is shared with the “=” initialization syntax, but not with parentheses:

```
class Widget {
  …

private:
  int x{ 0 };                 // fine, x's default value is 0
  int y = 0;                  // also fine
  int z(0);                   // error!
};
```

On the other hand, uncopyable objects (e.g., `std::atomics`—see Item 40) may be initialized using braces or parentheses, but not using “=”:

```
std::atomic<int> ai1{ 0 };      // fine
```

```
std::atomic<int> ai2(0);       // fine

std::atomic<int> ai3 = 0;      // error!
```

It's thus easy to understand why braced initialization is called "uniform." Of C++'s three ways to designate an initializing expression, only braces can be used everywhere.

A novel feature of braced initialization is that it prohibits implicit *narrowing conversions* among built-in types. If the value of an expression in a braced initializer isn't guaranteed to be expressible by the type of the object being initialized, the code won't compile:

```
double x, y, z;

…

int sum1{ x + y + z };        // error! sum of doubles may
                              // not be expressible as int
```

Initialization using parentheses and "=" doesn't check for narrowing conversions, because that could break too much legacy code:

```
int sum2(x + y + z);          // okay (value of expression
                              // truncated to an int)

int sum3 = x + y + z;         // ditto
```

Another noteworthy characteristic of braced initialization is its immunity to C++'s *most vexing parse*. A side effect of C++'s rule that anything that can be parsed as a declaration must be interpreted as one, the most vexing parse most frequently afflicts developers when they want to default-construct an object, but inadvertently end up declaring a function instead. The root of the problem is that if you want to call a constructor with an argument, you can do it like this,

```
Widget w1(10);      // call Widget ctor with argument 10
```

but if you try to call a `Widget` constructor with zero arguments using the analogous syntax, you declare a function instead of an object:

```
Widget w2();        // most vexing parse! declares a function
                    // named w2 that returns a Widget!
```

Functions can't be declared using braces for the parameter list, so default-constructing an object using braces doesn't have this problem:

```
Widget w3{};        // calls Widget ctor with no args
```

There's thus a lot to be said for braced initialization. It's the syntax that can be used in the widest variety of contexts, it prevents implicit narrowing conversions, and it's immune to C++'s most vexing parse. A trifecta of goodness! So why isn't this Item entitled something like "Prefer braced initialization syntax"?

The drawback to braced initialization is the sometimes-surprising behavior that accompanies it. Such behavior grows out of the unusually tangled relationship among braced initializers, `std::initializer_lists`, and constructor overload resolution. Their interactions can lead to code that seems like it should do one thing, but actually does another. For example, Item 2 explains that when an `auto`-declared variable has a braced initializer, the type deduced is `std::initializer_list`, even though other ways of declaring a variable with the same initializer would yield a more intuitive type. As a result, the more you like `auto`, the less enthusiastic you're likely to be about braced initialization.

In constructor calls, parentheses and braces have the same meaning as long as `std::initializer_list` parameters are not involved:

```
class Widget {
public:
  Widget(int i, bool b);      // ctors not declaring
  Widget(int i, double d);    // std::initializer_list params
  …
};

Widget w1(10, true);          // calls first ctor

Widget w2{10, true};          // also calls first ctor

Widget w3(10, 5.0);           // calls second ctor

Widget w4{10, 5.0};           // also calls second ctor
```

If, however, one or more constructors declare a parameter of type `std::initializer_list`, calls using the braced initialization syntax strongly prefer the overloads taking `std::initializer_lists`. *Strongly*. If there's *any way* for compilers to construe a call using a braced initializer to be to a constructor taking a `std::initializer_list`, compilers will employ that interpretation. If the `Widget` class above is augmented with a constructor taking a `std::initializer_list<long double>`, for example,

```
class Widget {
public:
  Widget(int i, bool b);                      // as before
  Widget(int i, double d);                    // as before
```

```
  Widget(std::initializer_list<long double> il);   // added

  …
};
```

Widgets w2 and w4 will be constructed using the new constructor, even though the type of the std::initializer_list elements (long double) is, compared to the non-std::initializer_list constructors, a worse match for both arguments! Look:

```
Widget w1(10, true);      // uses parens and, as before,
                          // calls first ctor

Widget w2{10, true};      // uses braces, but now calls
                          // std::initializer_list ctor
                          // (10 and true convert to long double)

Widget w3(10, 5.0);       // uses parens and, as before,
                          // calls second ctor

Widget w4{10, 5.0};       // uses braces, but now calls
                          // std::initializer_list ctor
                          // (10 and 5.0 convert to long double)
```

Even what would normally be copy and move construction can be hijacked by std::initializer_list constructors:

```
class Widget {
public:
  Widget(int i, bool b);                           // as before
  Widget(int i, double d);                         // as before
  Widget(std::initializer_list<long double> il);   // as before

  operator float() const;                          // convert
  …                                                // to float

};

Widget w5(w4);               // uses parens, calls copy ctor

Widget w6{w4};               // uses braces, calls
                             // std::initializer_list ctor
                             // (w4 converts to float, and float
                             // converts to long double)
```

```
Widget w7(std::move(w4));     // uses parens, calls move ctor

Widget w8{std::move(w4)};     // uses braces, calls
                              // std::initializer_list ctor
                              // (for same reason as w6)
```

Compilers' determination to match braced initializers with constructors taking `std::initializer_lists` is so strong, it prevails even if the best-match `std::initializer_list` constructor can't be called. For example:

```
class Widget {
public:
  Widget(int i, bool b);                       // as before
  Widget(int i, double d);                     // as before

  Widget(std::initializer_list<bool> il);      // element type is
                                               // now bool

  …                                            // no implicit
};                                             // conversion funcs

Widget w{10, 5.0};        // error! requires narrowing conversions
```

Here, compilers will ignore the first two constructors (the second of which offers an exact match on both argument types) and try to call the constructor taking a `std::initializer_list<bool>`. Calling that constructor would require converting an `int` (`10`) and a `double` (`5.0`) to `bool`s. Both conversions would be narrowing (`bool` can't exactly represent either value), and narrowing conversions are prohibited inside braced initializers, so the call is invalid, and the code is rejected.

Only if there's no way to convert the types of the arguments in a braced initializer to the type in a `std::initializer_list` do compilers fall back on normal overload resolution. For example, if we replace the `std::initializer_list<bool>` constructor with one taking a `std::initializer_list<std::string>`, the non-`std::initializer_list` constructors become candidates again, because there is no way to convert `int`s and `bool`s to `std::string`s:

```
class Widget {
public:
  Widget(int i, bool b);                   // as before
  Widget(int i, double d);                 // as before

  // std::initializer_list element type is now std::string
  Widget(std::initializer_list<std::string> il);
  …                                        // no implicit
```

```
};                                      // conversion funcs

Widget w1(10, true);      // uses parens, still calls first ctor

Widget w2{10, true};      // uses braces, now calls first ctor

Widget w3(10, 5.0);       // uses parens, still calls second ctor

Widget w4{10, 5.0};       // uses braces, now calls second ctor
```

This brings us near the end of our examination of braced initializers and constructor overloading, but there's an interesting edge case that needs to be addressed. Suppose you use an empty set of braces to construct an object that supports default construction and also supports `std::initializer_list` construction. What do your empty braces mean? If they mean "no arguments," you get default construction, but if they mean "empty `std::initializer_list`," you get construction from a `std::initializer_list` with no elements.

The rule is that you get default construction. Empty braces mean no arguments, not an empty `std::initializer_list`:

```
class Widget {
public:
  Widget();                                 // default ctor

  Widget(std::initializer_list<int> il);    // std::initializer
                                            // _list ctor

  …                                         // no implicit
};                                          // conversion funcs

Widget w1;             // calls default ctor

Widget w2{};           // also calls default ctor

Widget w3();           // most vexing parse! declares a function!
```

If you *want* to call a `std::initializer_list` constructor with an empty `std::initializer_list`, you do it by making the empty braces a constructor argument—by putting the empty braces inside the parentheses or braces demarcating what you're passing:

```
Widget w4({});         // calls std::initializer_list ctor
                       // with empty list

Widget w5{{}};         // ditto
```

At this point, with seemingly arcane rules about braced initializers, `std::initializer_lists`, and constructor overloading burbling about in your brain, you may be wondering how much of this information matters in day-to-day programming. More than you might think, because one of the classes directly affected is `std::vector`. `std::vector` has a non-`std::initializer_list` constructor that allows you to specify the initial size of the container and a value each of the initial elements should have, but it also has a constructor taking a `std::initializer_list` that permits you to specify the initial values in the container. If you create a `std::vector` of a numeric type (e.g., a `std::vector<int>`) and you pass two arguments to the constructor, whether you enclose those arguments in parentheses or braces makes a tremendous difference:

```
std::vector<int> v1(10, 20);  // use non-std::initializer_list
                              // ctor: create 10-element
                              // std::vector, all elements have
                              // value of 20

std::vector<int> v2{10, 20};  // use std::initializer_list ctor:
                              // create 2-element std::vector,
                              // element values are 10 and 20
```

But let's step back from `std::vector` and also from the details of parentheses, braces, and constructor overloading resolution rules. There are two primary takeaways from this discussion. First, as a class author, you need to be aware that if your set of overloaded constructors includes one or more functions taking a `std::initializer_list`, client code using braced initialization may see only the `std::initializer_list` overloads. As a result, it's best to design your constructors so that the overload called isn't affected by whether clients use parentheses or braces. In other words, learn from what is now viewed as an error in the design of the `std::vector` interface, and design your classes to avoid it.

An implication is that if you have a class with no `std::initializer_list` constructor, and you add one, client code using braced initialization may find that calls that used to resolve to non-`std::initializer_list` constructors now resolve to the new function. Of course, this kind of thing can happen any time you add a new function to a set of overloads: calls that used to resolve to one of the old overloads might start calling the new one. The difference with `std::initializer_list` constructor overloads is that a `std::initializer_list` overload doesn't just compete with other overloads, it overshadows them to the point where the other overloads may hardly be considered. So add such overloads only with great deliberation.

The second lesson is that as a class client, you must choose carefully between parentheses and braces when creating objects. Most developers end up choosing one kind of delimiter as a default, using the other only when they have to. Braces-by-default folks are attracted by their unrivaled breadth of applicability, their prohibition of narrowing conversions, and their immunity to C++'s most vexing parse. Such folks understand that in some cases (e.g., creation of a `std::vector` with a given size and initial element value), parentheses are required. On the other hand, the go-parentheses-go crowd embraces parentheses as their default argument delimiter. They're attracted to its consistency with the C++98 syntactic tradition, its avoidance of the `auto`-deduced-a-`std::initializer_list` problem, and the knowledge that their object creation calls won't be inadvertently waylaid by `std::initializer_list` constructors. They concede that sometimes only braces will do (e.g., when creating a container with particular values). There's no consensus that either approach is better than the other, so my advice is to pick one and apply it consistently.

If you're a template author, the tension between parentheses and braces for object creation can be especially frustrating, because, in general, it's not possible to know which should be used. For example, suppose you'd like to create an object of an arbitrary type from an arbitrary number of arguments. A variadic template makes this conceptually straightforward:

```
template<typename T,                  // type of object to create
         typename... Ts>              // types of arguments to use
void doSomeWork(Ts&&... params)
{

  create local T object from params...
  …

}
```

There are two ways to turn the line of pseudocode into real code (see Item 25 for information about `std::forward`):

```
T localObject(std::forward<Ts>(params)...);     // using parens

T localObject{std::forward<Ts>(params)...};     // using braces
```

So consider this calling code:

```
std::vector<int> v;
…
doSomeWork<std::vector<int>>(10, 20);
```

If `doSomeWork` uses parentheses when creating `localObject`, the result is a `std::vector` with 10 elements. If `doSomeWork` uses braces, the result is a `std::vector` with 2 elements. Which is correct? The author of `doSomeWork` can't know. Only the caller can.

This is precisely the problem faced by the Standard Library functions `std::make_unique` and `std::make_shared` (see Item 21). These functions resolve the problem by internally using parentheses and by documenting this decision as part of their interfaces.[1]

Things to Remember

- Braced initialization is the most widely usable initialization syntax, it prevents narrowing conversions, and it's immune to C++'s most vexing parse.
- During constructor overload resolution, braced initializers are matched to `std::initializer_list` parameters if at all possible, even if other constructors offer seemingly better matches.
- An example of where the choice between parentheses and braces can make a significant difference is creating a `std::vector<`*`numeric type`*`>` with two arguments.
- Choosing between parentheses and braces for object creation inside templates can be challenging.

Item 8: Prefer `nullptr` to `0` and `NULL`.

So here's the deal: the literal `0` is an `int`, not a pointer. If C++ finds itself looking at `0` in a context where only a pointer can be used, it'll grudgingly interpret `0` as a null pointer, but that's a fallback position. C++'s primary policy is that `0` is an `int`, not a pointer.

Practically speaking, the same is true of `NULL`. There is some uncertainty in the details in `NULL`'s case, because implementations are allowed to give `NULL` an integral type other than `int` (e.g., `long`). That's not common, but it doesn't really matter, because the issue here isn't the exact type of `NULL`, it's that neither `0` nor `NULL` has a pointer type.

1 More flexible designs—ones that permit callers to determine whether parentheses or braces should be used in functions generated from a template—are possible. For details, see the 5 June 2013 entry of *Andrzej's C++ blog*, "Intuitive interface — Part I."

In C++98, the primary implication of this was that overloading on pointer and integral types could lead to surprises. Passing `0` or `NULL` to such overloads never called a pointer overload:

```
void f(int);           // three overloads of f
void f(bool);
void f(void*);

f(0);                  // calls f(int), not f(void*)

f(NULL);               // might not compile, but typically calls
                       // f(int). Never calls f(void*)
```

The uncertainty regarding the behavior of `f(NULL)` is a reflection of the leeway granted to implementations regarding the type of `NULL`. If `NULL` is defined to be, say, `0L` (i.e., `0` as a `long`), the call is ambiguous, because conversion from `long` to `int`, `long` to `bool`, and `0L` to `void*` are considered equally good. The interesting thing about that call is the contradiction between the *apparent* meaning of the source code ("I'm calling `f` with `NULL`—the null pointer") and its *actual* meaning ("I'm calling `f` with some kind of integer—not the null pointer"). This counterintuitive behavior is what led to the guideline for C++98 programmers to avoid overloading on pointer and integral types. That guideline remains valid in C++11, because, the advice of this Item notwithstanding, it's likely that some developers will continue to use `0` and `NULL`, even though `nullptr` is a better choice.

`nullptr`'s advantage is that it doesn't have an integral type. To be honest, it doesn't have a pointer type, either, but you can think of it as a pointer of *all* types. `nullptr`'s actual type is `std::nullptr_t`, and, in a wonderfully circular definition, `std::nullptr_t` is defined to be the type of `nullptr`. The type `std::nullptr_t` implicitly converts to all raw pointer types, and that's what makes `nullptr` act as if it were a pointer of all types.

Calling the overloaded function `f` with `nullptr` calls the `void*` overload (i.e., the pointer overload), because `nullptr` can't be viewed as anything integral:

```
f(nullptr);            // calls f(void*) overload
```

Using `nullptr` instead of `0` or `NULL` thus avoids overload resolution surprises, but that's not its only advantage. It can also improve code clarity, especially when `auto` variables are involved. For example, suppose you encounter this in a code base:

```
auto result = findRecord( /* arguments */ );

if (result == 0) {
```

```
  …
}
```

If you don't happen to know (or can't easily find out) what `findRecord` returns, it may not be clear whether `result` is a pointer type or an integral type. After all, `0` (what `result` is tested against) could go either way. If you see the following, on the other hand,

```
auto result = findRecord( /* arguments */ );

if (result == nullptr) {
  …
}
```

there's no ambiguity: `result` must be a pointer type.

`nullptr` shines especially brightly when templates enter the picture. Suppose you have some functions that should be called only when the appropriate mutex has been locked. Each function takes a different kind of pointer:

```
int    f1(std::shared_ptr<Widget> spw);  // call these only when
double f2(std::unique_ptr<Widget> upw);  // the appropriate
bool   f3(Widget* pw);                   // mutex is locked
```

Calling code that wants to pass null pointers could look like this:

```
std::mutex f1m, f2m, f3m;         // mutexes for f1, f2, and f3

using MuxGuard =                  // C++11 typedef; see Item 9
  std::lock_guard<std::mutex>;
…

{
  MuxGuard g(f1m);              // lock mutex for f1
  auto result = f1(0);          // pass 0 as null ptr to f1
}                               // unlock mutex

…

{
  MuxGuard g(f2m);              // lock mutex for f2
  auto result = f2(NULL);       // pass NULL as null ptr to f2
}                               // unlock mutex

…

{
```

```
  MuxGuard g(f3m);              // lock mutex for f3
  auto result = f3(nullptr);    // pass nullptr as null ptr to f3
}                               // unlock mutex
```

The failure to use `nullptr` in the first two calls in this code is sad, but the code works, and that counts for something. However, the repeated pattern in the calling code—lock mutex, call function, unlock mutex—is more than sad. It's disturbing. This kind of source code duplication is one of the things that templates are designed to avoid, so let's templatize the pattern:

```
template<typename FuncType,
         typename MuxType,
         typename PtrType>
auto lockAndCall(FuncType func,
                 MuxType& mutex,
                 PtrType ptr) -> decltype(func(ptr))
{
  MuxGuard g(mutex);
  return func(ptr);
}
```

If the return type of this function (`auto … -> decltype(func(ptr))`) has you scratching your head, do your head a favor and navigate to Item 3, which explains what's going on. There you'll see that in C++14, the return type could be reduced to a simple `decltype(auto)`:

```
template<typename FuncType,
         typename MuxType,
         typename PtrType>
decltype(auto) lockAndCall(FuncType func,       // C++14
                           MuxType& mutex,
                           PtrType ptr)
{
  MuxGuard g(mutex);
  return func(ptr);
}
```

Given the `lockAndCall` template (either version), callers can write code like this:

```
auto result1 = lockAndCall(f1, f1m, 0);         // error!

…

auto result2 = lockAndCall(f2, f2m, NULL);      // error!

…
```

```
auto result3 = lockAndCall(f3, f3m, nullptr);    // fine
```

Well, they can write it, but, as the comments indicate, in two of the three cases, the code won't compile. The problem in the first call is that when `0` is passed to `lockAndCall`, template type deduction kicks in to figure out its type. The type of `0` is, was, and always will be `int`, so that's the type of the parameter `ptr` inside the instantiation of this call to `lockAndCall`. Unfortunately, this means that in the call to `func` inside `lockAndCall`, an `int` is being passed, and that's not compatible with the `std::shared_ptr<Widget>` parameter that `f1` expects. The `0` passed in the call to `lockAndCall` was intended to represent a null pointer, but what actually got passed was a run-of-the-mill `int`. Trying to pass this `int` to `f1` as a `std::shared_ptr<Widget>` is a type error. The call to `lockAndCall` with `0` fails because inside the template, an `int` is being passed to a function that requires a `std::shared_ptr<Widget>`.

The analysis for the call involving `NULL` is essentially the same. When `NULL` is passed to `lockAndCall`, an integral type is deduced for the parameter `ptr`, and a type error occurs when `ptr`—an `int` or `int`-like type—is passed to `f2`, which expects to get a `std::unique_ptr<Widget>`.

In contrast, the call involving `nullptr` has no trouble. When `nullptr` is passed to `lockAndCall`, the type for `ptr` is deduced to be `std::nullptr_t`. When `ptr` is passed to `f3`, there's an implicit conversion from `std::nullptr_t` to `Widget*`, because `std::nullptr_t` implicitly converts to all pointer types.

The fact that template type deduction deduces the "wrong" types for `0` and `NULL` (i.e., their true types, rather than their fallback meaning as a representation for a null pointer) is the most compelling reason to use `nullptr` instead of `0` or `NULL` when you want to refer to a null pointer. With `nullptr`, templates pose no special challenge. Combined with the fact that `nullptr` doesn't suffer from the overload resolution surprises that `0` and `NULL` are susceptible to, the case is ironclad. When you want to refer to a null pointer, use `nullptr`, not `0` or `NULL`.

Things to Remember

- Prefer `nullptr` to `0` and `NULL`.
- Avoid overloading on integral and pointer types.

Item 9: Prefer alias declarations to typedefs.

I'm confident we can agree that using STL containers is a good idea, and I hope that Item 18 convinces you that using `std::unique_ptr` is a good idea, but my guess is that neither of us is fond of writing types like "`std::unique_ptr<std::unordered_map<std::string, std::string>>`" more than once. Just thinking about it probably increases the risk of carpal tunnel syndrome.

Avoiding such medical tragedies is easy. Introduce a `typedef`:

```
typedef
  std::unique_ptr<std::unordered_map<std::string, std::string>>
  UPtrMapSS;
```

But `typedef`s are soooo C++98. They work in C++11, sure, but C++11 also offers *alias declarations*:

```
using UPtrMapSS =
  std::unique_ptr<std::unordered_map<std::string, std::string>>;
```

Given that the `typedef` and the alias declaration do exactly the same thing, it's reasonable to wonder whether there is a solid technical reason for preferring one over the other.

There is, but before I get to it, I want to mention that many people find the alias declaration easier to swallow when dealing with types involving function pointers:

```
// FP is a synonym for a pointer to a function taking an int and
// a const std::string& and returning nothing
typedef void (*FP)(int, const std::string&);      // typedef

// same meaning as above
using FP = void (*)(int, const std::string&);     // alias
                                                  // declaration
```

Of course, neither form is particularly easy to choke down, and few people spend much time dealing with synonyms for function pointer types, anyway, so this is hardly a compelling reason to choose alias declarations over `typedef`s.

But a compelling reason does exist: templates. In particular, alias declarations may be templatized (in which case they're called *alias templates*), while `typedef`s cannot. This gives C++11 programmers a straightforward mechanism for expressing things that in C++98 had to be hacked together with `typedef`s nested inside templatized `struct`s. For example, consider defining a synonym for a linked list that uses a custom allocator, `MyAlloc`. With an alias template, it's a piece of cake:

```
template<typename T>                          // MyAllocList<T>
using MyAllocList = std::list<T, MyAlloc<T>>; // is synonym for
                                              // std::list<T,
                                              //   MyAlloc<T>>

MyAllocList<Widget> lw;                       // client code
```

With a typedef, you pretty much have to create the cake from scratch:

```
template<typename T>                       // MyAllocList<T>::type
struct MyAllocList {                       // is synonym for
  typedef std::list<T, MyAlloc<T>> type;   // std::list<T,
};                                         //   MyAlloc<T>>

MyAllocList<Widget>::type lw;              // client code
```

It gets worse. If you want to use the typedef inside a template for the purpose of creating a linked list holding objects of a type specified by a template parameter, you have to precede the typedef name with typename:

```
template<typename T>
class Widget {                           // Widget<T> contains
private:                                 // a MyAllocList<T>
  typename MyAllocList<T>::type list;    // as a data member
  …
};
```

Here, `MyAllocList<T>::type` refers to a type that's dependent on a template type parameter (T). `MyAllocList<T>::type` is thus a *dependent type*, and one of C++'s many endearing rules is that the names of dependent types must be preceded by typename.

If MyAllocList is defined as an alias template, this need for typename vanishes (as does the cumbersome "::type" suffix):

```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>; // as before

template<typename T>
class Widget {
private:
  MyAllocList<T> list;                        // no "typename",
  …                                           // no "::type"
};
```

To you, `MyAllocList<T>` (i.e., use of the alias template) may look just as dependent on the template parameter T as `MyAllocList<T>::type` (i.e., use of the nested type def), but you're not a compiler. When compilers process the Widget template and encounter the use of MyAllocList<T> (i.e., use of the alias template), they know that MyAllocList<T> is the name of a type, because MyAllocList is an alias template: it *must* name a type. MyAllocList<T> is thus a *non-dependent type*, and a typename specifier is neither required nor permitted.

When compilers see MyAllocList<T>::type (i.e., use of the nested typedef) in the Widget template, on the other hand, they can't know for sure that it names a type, because there might be a specialization of MyAllocList that they haven't yet seen where MyAllocList<T>::type refers to something other than a type. That sounds crazy, but don't blame compilers for this possibility. It's the humans who have been known to produce such code.

For example, some misguided soul may have concocted something like this:

```
class Wine { … };

template<>                          // MyAllocList specialization
class MyAllocList<Wine> {           // for when T is Wine
private:
  enum class WineType               // see Item 10 for info on
  { White, Red, Rose };             // "enum class"

  WineType type;                    // in this class, type is
  …                                 // a data member!
};
```

As you can see, MyAllocList<Wine>::type doesn't refer to a type. If Widget were to be instantiated with Wine, MyAllocList<T>::type inside the Widget template would refer to a data member, not a type. Inside the Widget template, then, whether MyAllocList<T>::type refers to a type is honestly dependent on what T is, and that's why compilers insist on your asserting that it is a type by preceding it with typename.

If you've done any template metaprogramming (TMP), you've almost certainly bumped up against the need to take template type parameters and create revised types from them. For example, given some type T, you might want to strip off any const- or reference-qualifiers that T contains, e.g., you might want to turn const std::string& into std::string. Or you might want to add const to a type or turn it into an lvalue reference, e.g., turn Widget into const Widget or into Widget&. (If you haven't done any TMP, that's too bad, because if you want to be a truly effective C++ programmer, you need to be familiar with at least the basics of this facet of C++. You can see examples of TMP in action, including the kinds of type transformations I just mentioned, in Items 23 and 27.)

C++11 gives you the tools to perform these kinds of transformations in the form of *type traits*, an assortment of templates inside the header `<type_traits>`. There are dozens of type traits in that header, and not all of them perform type transformations, but the ones that do offer a predictable interface. Given a type `T` to which you'd like to apply a transformation, the resulting type is `std::`*`transformation`*`<T>::type`. For example:

```
std::remove_const<T>::type            // yields T from const T

std::remove_reference<T>::type        // yields T from T& and T&&

std::add_lvalue_reference<T>::type    // yields T& from T
```

The comments merely summarize what these transformations do, so don't take them too literally. Before using them on a project, you'd look up the precise specifications, I know.

My motivation here isn't to give you a tutorial on type traits, anyway. Rather, note that application of these transformations entails writing "`::type`" at the end of each use. If you apply them to a type parameter inside a template (which is virtually always how you employ them in real code), you'd also have to precede each use with `typename`. The reason for both of these syntactic speed bumps is that the C++11 type traits are implemented as nested `typedef`s inside templatized `struct`s. That's right, they're implemented using the type synonym technology I've been trying to convince you is inferior to alias templates!

There's a historical reason for that, but we'll skip over it (it's dull, I promise), because the Standardization Committee belatedly recognized that alias templates are the better way to go, and they included such templates in C++14 for all the C++11 type transformations. The aliases have a common form: for each C++11 transformation `std::`*`transformation`*`<T>::type`, there's a corresponding C++14 alias template named `std::`*`transformation`*`_t`. Examples will clarify what I mean:

```
std::remove_const<T>::type            // C++11: const T → T
std::remove_const_t<T>                // C++14 equivalent

std::remove_reference<T>::type        // C++11: T&/T&& → T
std::remove_reference_t<T>            // C++14 equivalent

std::add_lvalue_reference<T>::type    // C++11: T → T&
std::add_lvalue_reference_t<T>        // C++14 equivalent
```

The C++11 constructs remain valid in C++14, but I don't know why you'd want to use them. Even if you don't have access to C++14, writing the alias templates yourself is child's play. Only C++11 language features are required, and even children can mimic a pattern, right? If you happen to have access to an electronic copy of the C++14 Standard, it's easier still, because all that's required is some copying and pasting. Here, I'll get you started:

```
template <class T>
using remove_const_t = typename remove_const<T>::type;

template <class T>
using remove_reference_t = typename remove_reference<T>::type;

template <class T>
using add_lvalue_reference_t =
  typename add_lvalue_reference<T>::type;
```

See? Couldn't be easier.

Things to Remember

- `typedef`s don't support templatization, but alias declarations do.
- Alias templates avoid the "`::type`" suffix and, in templates, the "`typename`" prefix often required to refer to `typedef`s.
- C++14 offers alias templates for all the C++11 type traits transformations.

Item 10: Prefer scoped `enum`s to unscoped `enum`s.

As a general rule, declaring a name inside curly braces limits the visibility of that name to the scope defined by the braces. Not so for the enumerators declared in C++98-style `enum`s. The names of such enumerators belong to the scope containing the `enum`, and that means that nothing else in that scope may have the same name:

```
enum Color { black, white, red };   // black, white, red are
                                    // in same scope as Color

auto white = false;                 // error! white already
                                    // declared in this scope
```

The fact that these enumerator names leak into the scope containing their `enum` definition gives rise to the official term for this kind of `enum`: *unscoped*. Their new C++11 counterparts, *scoped* `enum`s, don't leak names in this way:

```
enum class Color { black, white, red };  // black, white, red
                                         // are scoped to Color

auto white = false;               // fine, no other
```

```
                                // "white" in scope

Color c = white;                // error! no enumerator named
                                // "white" is in this scope

Color c = Color::white;         // fine

auto c = Color::white;          // also fine (and in accord
                                // with Item 5's advice)
```

Because scoped enums are declared via "enum class", they're sometimes referred to as *enum classes*.

The reduction in namespace pollution offered by scoped enums is reason enough to prefer them over their unscoped siblings, but scoped enums have a second compelling advantage: their enumerators are much more strongly typed. Enumerators for unscoped enums implicitly convert to integral types (and, from there, to floating-point types). Semantic travesties such as the following are therefore completely valid:

```
enum Color { black, white, red };       // unscoped enum

std::vector<std::size_t>                // func. returning
  primeFactors(std::size_t x);          // prime factors of x

Color c = red;
…

if (c < 14.5) {                     // compare Color to double (!)

  auto factors =                    // compute prime factors
    primeFactors(c);                // of a Color (!)
  …
}
```

Throw a simple "class" after "enum", however, thus transforming an unscoped enum into a scoped one, and it's a very different story. There are no implicit conversions from enumerators in a scoped enum to any other type:

```
enum class Color { black, white, red }; // enum is now scoped

Color c = Color::red;                   // as before, but
…                                       // with scope qualifier

if (c < 14.5) {                     // error! can't compare
                                    // Color and double
```

```
  auto factors =                  // error! can't pass Color to
    primeFactors(c);              // function expecting std::size_t
  …
}
```

If you honestly want to perform a conversion from `Color` to a different type, do what you always do to twist the type system to your wanton desires—use a cast:

```
if (static_cast<double>(c) < 14.5) {        // odd code, but
                                            // it's valid

  auto factors =                                  // suspect, but
    primeFactors(static_cast<std::size_t>(c));    // it compiles
  …
}
```

It may seem that scoped `enum`s have a third advantage over unscoped `enum`s, because scoped `enum`s may be forward-declared, i.e., their names may be declared without specifying their enumerators:

```
enum Color;                 // error!

enum class Color;           // fine
```

This is misleading. In C++11, unscoped `enum`s may also be forward-declared, but only after a bit of additional work. The work grows out of the fact that every `enum` in C++ has an integral *underlying type* that is determined by compilers. For an unscoped `enum` like `Color`,

```
enum Color { black, white, red };
```

compilers might choose `char` as the underlying type, because there are only three values to represent. However, some `enum`s have a range of values that is much larger, e.g.:

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
            };
```

Here the values to be represented range from `0` to `0xFFFFFFFF`. Except on unusual machines (where a `char` consists of at least 32 bits), compilers will have to select an integral type larger than `char` for the representation of `Status` values.

To make efficient use of memory, compilers often want to choose the smallest underlying type for an `enum` that's sufficient to represent its range of enumerator values. In some cases, compilers will optimize for speed instead of size, and in that case, they may not choose the smallest permissible underlying type, but they certainly want to be *able* to optimize for size. To make that possible, C++98 supports only `enum` definitions (where all enumerators are listed); `enum` declarations are not allowed. That makes it possible for compilers to select an underlying type for each `enum` prior to the `enum` being used.

But the inability to forward-declare `enum`s has drawbacks. The most notable is probably the increase in compilation dependencies. Consider again the `Status` `enum`:

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
            };
```

This is the kind of `enum` that's likely to be used throughout a system, hence included in a header file that every part of the system is dependent on. If a new status value is then introduced,

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              audited = 500,
              indeterminate = 0xFFFFFFFF
            };
```

it's likely that the entire system will have to be recompiled, even if only a single subsystem—possibly only a single function!—uses the new enumerator. This is the kind of thing that people *hate*. And it's the kind of thing that the ability to forward-declare `enum`s in C++11 eliminates. For example, here's a perfectly valid declaration of a scoped `enum` and a function that takes one as a parameter:

```
enum class Status;                   // forward declaration

void continueProcessing(Status s);   // use of fwd-declared enum
```

The header containing these declarations requires no recompilation if `Status`'s definition is revised. Furthermore, if `Status` is modified (e.g., to add the `audited` enumerator), but `continueProcessing`'s behavior is unaffected (e.g., because continueProcessing doesn't use audited), continueProcessing's implementation need not be recompiled, either.

But if compilers need to know the size of an enum before it's used, how can C++11's enums get away with forward declarations when C++98's enums can't? The answer is simple: the underlying type for a scoped enum is always known, and for unscoped enums, you can specify it.

By default, the underlying type for scoped enums is int:

```
enum class Status;                 // underlying type is int
```

If the default doesn't suit you, you can override it:

```
enum class Status: std::uint32_t;  // underlying type for
                                   // Status is std::uint32_t
                                   // (from <cstdint>)
```

Either way, compilers know the size of the enumerators in a scoped enum.

To specify the underlying type for an unscoped enum, you do the same thing as for a scoped enum, and the result may be forward-declared:

```
enum Color: std::uint8_t;        // fwd decl for unscoped enum;
                                 // underlying type is
                                 // std::uint8_t
```

Underlying type specifications can also go on an enum's definition:

```
enum class Status: std::uint32_t { good = 0,
                                   failed = 1,
                                   incomplete = 100,
                                   corrupt = 200,
                                   audited = 500,
                                   indeterminate = 0xFFFFFFFF
                                 };
```

In view of the fact that scoped enums avoid namespace pollution and aren't susceptible to nonsensical implicit type conversions, it may surprise you to hear that there's at least one situation where unscoped enums may be useful. That's when referring to fields within C++11's std::tuples. For example, suppose we have a tuple holding values for the name, email address, and reputation value for a user at a social networking website:

```
using UserInfo =                 // type alias; see Item 9
  std::tuple<std::string,        // name
```

```
                std::string,        // email
                std::size_t> ;      // reputation
```

Though the comments indicate what each field of the tuple represents, that's probably not very helpful when you encounter code like this in a separate source file:

```
UserInfo uInfo;                   // object of tuple type
…

auto val = std::get<1>(uInfo);    // get value of field 1
```

As a programmer, you have a lot of stuff to keep track of. Should you really be expected to remember that field 1 corresponds to the user's email address? I think not. Using an unscoped `enum` to associate names with field numbers avoids the need to:

```
enum UserInfoFields { uiName, uiEmail, uiReputation };

UserInfo uInfo;                         // as before
…

auto val = std::get<uiEmail>(uInfo);    // ah, get value of
                                        // email field
```

What makes this work is the implicit conversion from `UserInfoFields` to `std::size_t`, which is the type that `std::get` requires.

The corresponding code with scoped `enum`s is substantially more verbose:

```
enum class UserInfoFields { uiName, uiEmail, uiReputation };

UserInfo uInfo;                         // as before
…

auto val =
  std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>
    (uInfo);
```

The verbosity can be reduced by writing a function that takes an enumerator and returns its corresponding `std::size_t` value, but it's a bit tricky. `std::get` is a template, and the value you provide is a template argument (notice the use of angle brackets, not parentheses), so the function that transforms an enumerator into a `std::size_t` has to produce its result *during compilation*. As Item 15 explains, that means it must be a `constexpr` function.

In fact, it should really be a `constexpr` function template, because it should work with any kind of `enum`. And if we're going to make that generalization, we should generalize the return type, too. Rather than returning `std::size_t`, we'll return the `enum`'s underlying type. It's available via the `std::underlying_type` type trait. (See Item 9 for information on type traits.) Finally, we'll declare it `noexcept` (see Item 14), because we know it will never yield an exception. The result is a function template `toUType` that takes an arbitrary enumerator and can return its value as a compile-time constant:

```
template<typename E>
constexpr typename std::underlying_type<E>::type
  toUType(E enumerator) noexcept
{
  return
    static_cast<typename
                std::underlying_type<E>::type>(enumerator);
}
```

In C++14, `toUType` can be simplified by replacing `typename std::underlying_type<E>::type` with the sleeker `std::underlying_type_t` (see Item 9):

```
template<typename E>                              // C++14
constexpr std::underlying_type_t<E>
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

The even-sleeker `auto` return type (see Item 3) is also valid in C++14:

```
template<typename E>                              // C++14
constexpr auto
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

Regardless of how it's written, `toUType` permits us to access a field of the tuple like this:

```
auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

It's still more to write than use of the unscoped `enum`, but it also avoids namespace pollution and inadvertent conversions involving enumerators. In many cases, you may decide that typing a few extra characters is a reasonable price to pay for the ability to avoid the pitfalls of an enum technology that dates to a time when the state of the art in digital telecommunications was the 2400-baud modem.

Things to Remember

- C++98-style `enum`s are now known as unscoped `enum`s.
- Enumerators of scoped `enum`s are visible only within the `enum`. They convert to other types only with a cast.
- Both scoped and unscoped `enum`s support specification of the underlying type. The default underlying type for scoped `enum`s is `int`. Unscoped `enum`s have no default underlying type.
- Scoped `enum`s may always be forward-declared. Unscoped `enum`s may be forward-declared only if their declaration specifies an underlying type.

Item 11: Prefer deleted functions to private undefined ones.

If you're providing code to other developers, and you want to prevent them from calling a particular function, you generally just don't declare the function. No function declaration, no function to call. Easy, peasy. But sometimes C++ declares functions for you, and if you want to prevent clients from calling those functions, the peasy isn't quite so easy any more.

The situation arises only for the "special member functions," i.e., the member functions that C++ automatically generates when they're needed. Item 17 discusses these functions in detail, but for now, we'll worry only about the copy constructor and the copy assignment operator. This chapter is largely devoted to common practices in C++98 that have been superseded by better practices in C++11, and in C++98, if you want to suppress use of a member function, it's almost always the copy constructor, the assignment operator, or both.

The C++98 approach to preventing use of these functions is to declare them `private` and not define them. For example, near the base of the iostreams hierarchy in the C++ Standard Library is the class template `basic_ios`. All istream and ostream classes inherit (possibly indirectly) from this class. Copying istreams and ostreams is undesirable, because it's not really clear what such operations should do. An `istream` object, for example, represents a stream of input values, some of which may have already been read, and some of which will potentially be read later. If an istream were to be copied, would that entail copying all the values that had already been read as well as all the values that would be read in the future? The easiest way to deal with such questions is to define them out of existence. Prohibiting the copying of streams does just that.

To render istream and ostream classes uncopyable, `basic_ios` is specified in C++98 as follows (including the comments):

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
  …

private:
  basic_ios(const basic_ios& );              // not defined
  basic_ios& operator=(const basic_ios&);   // not defined
};
```

Declaring these functions `private` prevents clients from calling them. Deliberately failing to define them means that if code that still has access to them (i.e., member functions or `friend`s of the class) uses them, linking will fail due to missing function definitions.

In C++11, there's a better way to achieve essentially the same end: use "`= delete`" to mark the copy constructor and the copy assignment operator as *deleted functions*. Here's the same part of `basic_ios` as it's specified in C++11:

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
  …
  basic_ios(const basic_ios& ) = delete;
  basic_ios& operator=(const basic_ios&) = delete;
  …
};
```

The difference between deleting these functions and declaring them `private` may seem more a matter of fashion than anything else, but there's greater substance here than you might think. Deleted functions may not be used in any way, so even code that's in member and `friend` functions will fail to compile if it tries to copy `basic_ios` objects. That's an improvement over the C++98 behavior, where such improper usage wouldn't be diagnosed until link-time.

By convention, deleted functions are declared `public`, not `private`. There's a reason for that. When client code tries to use a member function, C++ checks accessibility before deleted status. When client code tries to use a deleted `private` function, some compilers complain only about the function being `private`, even though the function's accessibility doesn't really affect whether it can be used. It's worth bearing this in mind when revising legacy code to replace `private`-and-not-defined member functions with deleted ones, because making the new functions `public` will generally result in better error messages.

An important advantage of deleted functions is that *any* function may be deleted, while only member functions may be `private`. For example, suppose we have a non-member function that takes an integer and returns whether it's a lucky number:

```
bool isLucky(int number);
```

C++'s C heritage means that pretty much any type that can be viewed as vaguely numerical will implicitly convert to `int`, but some calls that would compile might not make sense:

```
if (isLucky('a')) …            // is 'a' a lucky number?

if (isLucky(true)) …           // is "true"?

if (isLucky(3.5)) …            // should we truncate to 3
                               // before checking for luckiness?
```

If lucky numbers must really be integers, we'd like to prevent calls such as these from compiling.

One way to accomplish that is to create deleted overloads for the types we want to filter out:

```
bool isLucky(int number);            // original function

bool isLucky(char) = delete;         // reject chars

bool isLucky(bool) = delete;         // reject bools

bool isLucky(double) = delete;       // reject doubles and
                                     // floats
```

(The comment on the `double` overload that says that both `double`s and `float`s will be rejected may surprise you, but your surprise will dissipate once you recall that, given a choice between converting a `float` to an `int` or to a `double`, C++ prefers the conversion to `double`. Calling `isLucky` with a `float` will therefore call the `double` overload, not the `int` one. Well, it'll try to. The fact that that overload is deleted will prevent the call from compiling.)

Although deleted functions can't be used, they are part of your program. As such, they are taken into account during overload resolution. That's why, with the deleted function declarations above, the undesirable calls to `isLucky` will be rejected:

```
if (isLucky('a')) …          // error! call to deleted function
```

```
if (isLucky(true)) …            // error!

if (isLucky(3.5f)) …            // error!
```

Another trick that deleted functions can perform (and that `private` member functions can't) is to prevent use of template instantiations that should be disabled. For example, suppose you need a template that works with built-in pointers (Chapter 4's advice to prefer smart pointers to raw pointers notwithstanding):

```
template<typename T>
void processPointer(T* ptr);
```

There are two special cases in the world of pointers. One is `void*` pointers, because there is no way to dereference them, to increment or decrement them, etc. The other is `char*` pointers, because they typically represent pointers to C-style strings, not pointers to individual characters. These special cases often call for special handling, and, in the case of the `processPointer` template, let's assume the proper handling is to reject calls using those types. That is, it should not be possible to call `processPointer` with `void*` or `char*` pointers.

That's easily enforced. Just delete those instantiations:

```
template<>
void processPointer<void>(void*) = delete;

template<>
void processPointer<char>(char*) = delete;
```

Now, if calling `processPointer` with a `void*` or a `char*` is invalid, it's probably also invalid to call it with a `const void*` or a `const char*`, so those instantiations will typically need to be deleted, too:

```
template<>
void processPointer<const void>(const void*) = delete;

template<>
void processPointer<const char>(const char*) = delete;
```

And if you really want to be thorough, you'll also delete the `const volatile void*` and `const volatile char*` overloads, and then you'll get to work on the overloads for pointers to the other standard character types: `std::wchar_t`, `std::char16_t`, and `std::char32_t`.

Interestingly, if you have a function template inside a class, and you'd like to disable some instantiations by declaring them `private` (à la classic C++98 convention), you can't, because it's not possible to give a member function template specialization a different access level from that of the main template. If `processPointer` were a member function template inside `Widget`, for example, and you wanted to disable calls for `void*` pointers, this would be the C++98 approach, though it would not compile:

```
class Widget {
public:
  …
  template<typename T>
  void processPointer(T* ptr)
  { … }

private:
  template<>                                 // error!
  void processPointer<void>(void*);

};
```

The problem is that template specializations must be written at namespace scope, not class scope. This issue doesn't arise for deleted functions, because they don't need a different access level. They can be deleted outside the class (hence at namespace scope):

```
class Widget {
public:
  …
  template<typename T>
  void processPointer(T* ptr)
  { … }
  …

};

template<>                                          // still
void Widget::processPointer<void>(void*) = delete;  // public,
                                                    // but
                                                    // deleted
```

The truth is that the C++98 practice of declaring functions `private` and not defining them was really an attempt to achieve what C++11's deleted functions actually accomplish. As an emulation, the C++98 approach is not as good as the real thing. It doesn't work outside classes, it doesn't always work inside classes, and when it does work, it may not work until link-time. So stick to deleted functions.

Things to Remember

- Prefer deleted functions to private undefined ones.
- Any function may be deleted, including non-member functions and template instantiations.

Item 12: Declare overriding functions `override`.

The world of object-oriented programming in C++ revolves around classes, inheritance, and virtual functions. Among the most fundamental ideas in this world is that virtual function implementations in derived classes *override* the implementations of their base class counterparts. It's disheartening, then, to realize just how easily virtual function overriding can go wrong. It's almost as if this part of the language were designed with the idea that Murphy's Law wasn't just to be obeyed, it was to be honored.

Because "overriding" sounds a lot like "overloading," yet is completely unrelated, let me make clear that virtual function overriding is what makes it possible to invoke a derived class function through a base class interface:

```
class Base {
public:
  virtual void doWork();         // base class virtual function
  …
};

class Derived: public Base {
public:
  virtual void doWork();         // overrides Base::doWork
  …                              // ("virtual" is optional
};                               // here)

std::unique_ptr<Base> upb =      // create base class pointer
  std::make_unique<Derived>();   // to derived class object;
                                 // see Item 21 for info on
…                                // std::make_unique

upb->doWork();                   // call doWork through base
                                 // class ptr; derived class
                                 // function is invoked
```

For overriding to occur, several requirements must be met:

- The base class function must be virtual.
- The base and derived function names must be identical (except in the case of destructors).
- The parameter types of the base and derived functions must be identical.
- The `const`ness of the base and derived functions must be identical.
- The return types and exception specifications of the base and derived functions must be compatible.

To these constraints, which were also part of C++98, C++11 adds one more:

- The functions' *reference qualifiers* must be identical. Member function reference qualifiers are one of C++11's less-publicized features, so don't be surprised if you've never heard of them. They make it possible to limit use of a member function to lvalues only or to rvalues only. Member functions need not be virtual to use them:

```
class Widget {
public:
  …
  void doWork() &;         // this version of doWork applies
                           // only when *this is an lvalue

  void doWork() &&;        // this version of doWork applies
};                         // only when *this is an rvalue

…

Widget makeWidget();       // factory function (returns rvalue)

Widget w;                  // normal object (an lvalue)

…

w.doWork();                // calls Widget::doWork for lvalues
                           // (i.e., Widget::doWork &)

makeWidget().doWork();     // calls Widget::doWork for rvalues
                           // (i.e., Widget::doWork &&)
```

  I'll say more about member functions with reference qualifiers later, but for now, simply note that if a virtual function in a base class has a reference qualifier, derived class overrides of that function must have exactly the same reference qualifier. If they don't, the declared functions will still exist in the derived class, but they won't override anything in the base class.

All these requirements for overriding mean that small mistakes can make a big difference. Code containing overriding errors is typically valid, but its meaning isn't what you intended. You therefore can't rely on compilers notifying you if you do something wrong. For example, the following code is completely legal and, at first sight, looks reasonable, but it contains no virtual function overrides—not a single derived class function that is tied to a base class function. Can you identify the problem in each case, i.e., why each derived class function doesn't override the base class function with the same name?

```
class Base {
public:
  virtual void mf1() const;
  virtual void mf2(int x);
  virtual void mf3() &;
  void mf4() const;
};

class Derived: public Base {
public:
  virtual void mf1();
  virtual void mf2(unsigned int x);
  virtual void mf3() &&;
  void mf4() const;
};
```

Need some help?

- `mf1` is declared `const` in `Base`, but not in `Derived`.
- `mf2` takes an `int` in `Base`, but an `unsigned int` in `Derived`.
- `mf3` is lvalue-qualified in `Base`, but rvalue-qualified in `Derived`.
- `mf4` isn't declared `virtual` in `Base`.

You may think, "Hey, in practice, these things will elicit compiler warnings, so I don't need to worry." Maybe that's true. But maybe it's not. With two of the compilers I checked, the code was accepted without complaint, and that was with all warnings enabled. (Other compilers provided warnings about some of the issues, but not all of them.)

Because declaring derived class overrides is important to get right, but easy to get wrong, C++11 gives you a way to make explicit that a derived class function is supposed to override a base class version: declare it `override`. Applying this to the example above would yield this derived class:

```
class Derived: public Base {
public:
  virtual void mf1() override;
  virtual void mf2(unsigned int x) override;
  virtual void mf3() && override;
  virtual void mf4() const override;
};
```

This won't compile, of course, because when written this way, compilers will kvetch about all the overriding-related problems. That's exactly what you want, and it's why you should declare all your overriding functions `override`.

The code using `override` that does compile looks as follows (assuming that the goal is for all functions in `Derived` to override virtuals in `Base`):

```
class Base {
public:
  virtual void mf1() const;
  virtual void mf2(int x);
  virtual void mf3() &;
  virtual void mf4() const;
};

class Derived: public Base {
public:
  virtual void mf1() const override;
  virtual void mf2(int x) override;
  virtual void mf3() & override;
  void mf4() const override;          // adding "virtual" is OK,
};                                    // but not necessary
```

Note that in this example, part of getting things to work involves declaring `mf4` virtual in `Base`. Most overriding-related errors occur in derived classes, but it's possible for things to be incorrect in base classes, too.

A policy of using `override` on all your derived class overrides can do more than just enable compilers to tell you when would-be overrides aren't overriding anything. It can also help you gauge the ramifications if you're contemplating changing the signature of a virtual function in a base class. If derived classes use `override` everywhere, you can just change the signature, recompile your system, see how much damage you've caused (i.e., how many derived classes fail to compile), then decide whether the signature change is worth the trouble. Without `override`, you'd have to hope you have comprehensive unit tests in place, because, as we've seen, derived class virtuals that are supposed to override base class functions, but don't, need not elicit compiler diagnostics.

C++ has always had keywords, but C++11 introduces two *contextual keywords*, `override` and `final`.[2] These keywords have the characteristic that they are reserved, but only in certain contexts. In the case of `override`, it has a reserved meaning only when it occurs at the end of a member function declaration. That means that if you have legacy code that already uses the name `override`, you don't need to change it for C++11:

```
class Warning {              // potential legacy class from C++98
public:
  …
  void override();           // legal in both C++98 and C++11
  …                          // (with the same meaning)
};
```

That's all there is to say about `override`, but it's not all there is to say about member function reference qualifiers. I promised I'd provide more information on them later, and now it's later.

If we want to write a function that accepts only lvalue arguments, we declare a non-`const` lvalue reference parameter:

```
void doSomething(Widget& w);      // accepts only lvalue Widgets
```

If we want to write a function that accepts only rvalue arguments, we declare an rvalue reference parameter:

```
void doSomething(Widget&& w);     // accepts only rvalue Widgets
```

Member function reference qualifiers simply make it possible to draw the same distinction for the object on which a member function is invoked, i.e., `*this`. It's precisely analogous to the `const` at the end of a member function declaration, which indicates that the object on which the member function is invoked (i.e., `*this`) is `const`.

The need for reference-qualified member functions is not common, but it can arise. For example, suppose our `Widget` class has a `std::vector` data member, and we offer an accessor function that gives clients direct access to it:

```
class Widget {
public:
```

2 Applying `final` to a virtual function prevents the function from being overridden in derived classes. `final` may also be applied to a class, in which case the class is prohibited from being used as a base class.

```
  using DataType = std::vector<double>;      // see Item 9 for
  …                                          // info on "using"

  DataType& data() { return values; }
  …

private:
  DataType values;
};
```

This is hardly the most encapsulated design that's seen the light of day, but set that aside and consider what happens in this client code:

```
Widget w;
…

auto vals1 = w.data();             // copy w.values into vals1
```

The return type of `Widget::data` is an lvalue reference (a `std::vector<double>&`, to be precise), and because lvalue references are defined to be lvalues, we're initializing `vals1` from an lvalue. `vals1` is thus copy constructed from `w.values`, just as the comment says.

Now suppose we have a factory function that creates `Widget`s,

```
Widget makeWidget();
```

and we want to initialize a variable with the `std::vector` inside the `Widget` returned from `makeWidget`:

```
auto vals2 = makeWidget().data();    // copy values inside the
                                     // Widget into vals2
```

Again, `Widgets::data` returns an lvalue reference, and, again, the lvalue reference is an lvalue, so, again, our new object (`vals2`) is copy constructed from `values` inside the `Widget`. This time, though, the `Widget` is the temporary object returned from `makeWidget` (i.e., an rvalue), so copying the `std::vector` inside it is a waste of time. It'd be preferable to move it, but, because `data` is returning an lvalue reference, the rules of C++ require that compilers generate code for a copy. (There's some wiggle room for optimization through what is known as the "as if rule," but you'd be foolish to rely on your compilers finding a way to take advantage of it.)

What's needed is a way to specify that when `data` is invoked on an rvalue `Widget`, the result should also be an rvalue. Using reference qualifiers to overload `data` for lvalue and rvalue `Widgets` makes that possible:

```
class Widget {
public:
  using DataType = std::vector<double>;
  …

  DataType& data() &                // for lvalue Widgets,
  { return values; }                // return lvalue

  DataType data() &&                // for rvalue Widgets,
  { return std::move(values); }     // return rvalue
  …

private:
  DataType values;
};
```

Notice the differing return types from the `data` overloads. The lvalue reference overload returns an lvalue reference (i.e., an lvalue), and the rvalue reference overload returns a temporary object (i.e., an rvalue). This means that client code now behaves as we'd like:

```
auto vals1 = w.data();               // calls lvalue overload for
                                     // Widget::data, copy-
                                     // constructs vals1

auto vals2 = makeWidget().data();   // calls rvalue overload for
                                     // Widget::data, move-
                                     // constructs vals2
```

This is certainly nice, but don't let the warm glow of this happy ending distract you from the true point of this Item. That point is that whenever you declare a function in a derived class that's meant to override a virtual function in a base class, be sure to declare that function `override`.

Things to Remember

- Declare overriding functions `override`.
- Member function reference qualifiers make it possible to treat lvalue and rvalue objects (`*this`) differently.

Item 13: Prefer `const_iterator`s to `iterator`s.

`const_iterator`s are the STL equivalent of pointers-to-`const`. They point to values that may not be modified. The standard practice of using `const` whenever possible dictates that you should use `const_iterator`s any time you need an iterator, yet have no need to modify what the iterator points to.

That's as true for C++98 as for C++11, but in C++98, `const_iterator`s had only halfhearted support. It wasn't that easy to create them, and once you had one, the ways you could use it were limited. For example, suppose you want to search a `std::vector<int>` for the first occurrence of 1983 (the year "C++" replaced "C with Classes" as the name of the programming language), then insert the value 1998 (the year the first ISO C++ Standard was adopted) at that location. If there's no 1983 in the vector, the insertion should go at the end of the vector. Using `iterator`s in C++98, that was easy:

```
std::vector<int> values;

…

std::vector<int>::iterator it =
  std::find(values.begin(),values.end(), 1983);
values.insert(it, 1998);
```

But `iterator`s aren't really the proper choice here, because this code never modifies what an `iterator` points to. Revising the code to use `const_iterator`s should be trivial, but in C++98, it was anything but. Here's one approach that's conceptually sound, though still not correct:

```
typedef std::vector<int>::iterator IterT;             // type-
typedef std::vector<int>::const_iterator ConstIterT;  // defs

std::vector<int> values;

…

ConstIterT ci =
  std::find(static_cast<ConstIterT>(values.begin()),  // cast
            static_cast<ConstIterT>(values.end()),    // cast
            1983);

values.insert(static_cast<IterT>(ci), 1998);    // may not
                                                // compile; see
                                                // below
```

The typedefs aren't required, of course, but they make the casts in the code easier to write. (If you're wondering why I'm showing typedefs instead of following the advice of Item 9 to use alias declarations, it's because this example shows C++98 code, and alias declarations are a feature new to C++11.)

The casts in the call to std::find are present because values is a non-const container and in C++98, there was no simple way to get a const_iterator from a non-const container. The casts aren't strictly necessary, because it was possible to get const_iterators in other ways (e.g., you could bind values to a reference-to-const variable, then use that variable in place of values in your code), but one way or another, the process of getting const_iterators to elements of a non-const container involved some amount of contorting.

Once you had the const_iterators, matters often got worse, because in C++98, locations for insertions (and erasures) could be specified only by iterators. const_iterators weren't acceptable. That's why, in the code above, I cast the const_iterator (that I was so careful to get from std::find) into an iterator: passing a const_iterator to insert wouldn't compile.

To be honest, the code I've shown might not compile, either, because there's no portable conversion from a const_iterator to an iterator, not even with a static_cast. Even the semantic sledgehammer known as reinterpret_cast can't do the job. (That's not a C++98 restriction. It's true in C++11, too. const_iterators simply don't convert to iterators, no matter how much it might seem like they should.) There are some portable ways to generate iterators that point where const_iterators do, but they're not obvious, not universally applicable, and not worth discussing in this book. Besides, I hope that by now my point is clear: const_iterators were so much trouble in C++98, they were rarely worth the bother. At the end of the day, developers don't use const whenever *possible*, they use it whenever *practical*, and in C++98, const_iterators just weren't very practical.

All that changed in C++11. Now const_iterators are both easy to get and easy to use. The container member functions cbegin and cend produce const_iterators, even for non-const containers, and STL member functions that use iterators to identify positions (e.g., insert and erase) actually use const_iterators. Revising the original C++98 code that uses iterators to use const_iterators in C++11 is truly trivial:

```
std::vector<int> values;                          // as before

…

auto it =                                         // use cbegin
```

```
    std::find(values.cbegin(),values.cend(), 1983);  // and cend

  values.insert(it, 1998);
```

Now *that's* code using `const_iterator`s that's practical!

About the only situation in which C++11's support for `const_iterator`s comes up a bit short is when you want to write maximally generic library code. Such code takes into account that some containers and container-like data structures offer `begin` and `end` (plus `cbegin`, `cend`, `rbegin`, etc.) as *non-member* functions, rather than members. This is the case for built-in arrays, for example, and it's also the case for some third-party libraries with interfaces consisting only of free functions. Maximally generic code thus uses non-member functions rather than assuming the existence of member versions.

For example, we could generalize the code we've been working with into a `findAndInsert` template as follows:

```
template<typename C, typename V>
void findAndInsert(C& container,          // in container, find
                   const V& targetVal,    // first occurrence
                   const V& insertVal)    // of targetVal, then
{                                         // insert insertVal
  using std::cbegin;                      // there
  using std::cend;

  auto it = std::find(cbegin(container),  // non-member cbegin
                      cend(container),    // non-member cend
                      targetVal);

  container.insert(it, insertVal);
}
```

This works fine in C++14, but, sadly, not in C++11. Through an oversight during standardization, C++11 added the non-member functions `begin` and `end`, but it failed to add `cbegin`, `cend`, `rbegin`, `rend`, `crbegin`, and `crend`. C++14 rectifies that oversight.

If you're using C++11, you want to write maximally generic code, and none of the libraries you're using provides the missing templates for non-member `cbegin` and friends, you can throw your own implementations together with ease. For example, here's an implementation of non-member `cbegin`:

```
template <class C>
auto cbegin(const C& container)->decltype(std::begin(container))
{
```

```
  return std::begin(container);         // see explanation below
}
```

You're surprised to see that non-member `cbegin` doesn't call member `cbegin`, aren't you? So was I. But follow the logic. This `cbegin` template accepts any type of argument representing a container-like data structure, `C`, and it accesses this argument through its reference-to-`const` parameter, `container`. If `C` is a conventional container type (e.g., a `std::vector<int>`), `container` will be a reference to a `const` version of that container (e.g., a `const std::vector<int>&`). Invoking the non-member `begin` function (provided by C++11) on a `const` container yields a `const_iterator`, and that iterator is what this template returns. The advantage of implementing things this way is that it works even for containers that offer a `begin` member function (which, for containers, is what C++11's non-member `begin` calls), but fail to offer a `cbegin` member. You can thus use this non-member `cbegin` on containers that directly support only `begin`.

This template also works if `C` is a built-in array type. In that case, `container` becomes a reference to a `const` array. C++11 provides a specialized version of non-member `begin` for arrays that returns a pointer to the array's first element. The elements of a `const` array are `const`, so the pointer that non-member `begin` returns for a `const` array is a pointer-to-`const`, and a pointer-to-`const` is, in fact, a `const_iterator` for an array. (For insight into how a template can be specialized for built-in arrays, consult Item 1's discussion of type deduction in templates that take reference parameters to arrays.)

But back to basics. The point of this Item is to encourage you to use `const_iterators` whenever you can. The fundamental motivation—using `const` whenever it's meaningful—predates C++11, but in C++98, it simply wasn't practical when working with iterators. In C++11, it's eminently practical, and C++14 tidies up the few bits of unfinished business that C++11 left behind.

Things to Remember

- Prefer `const_iterators` to `iterators`.
- In maximally generic code, prefer non-member versions of `begin`, `end`, `rbegin`, etc., over their member function counterparts.

Item 14: Declare functions `noexcept` if they won't emit exceptions.

In C++98, exception specifications were rather temperamental beasts. You had to summarize the exception types a function might emit, so if the function's implementation was modified, the exception specification might require revision, too. Changing an exception specification could break client code, because callers might be dependent on the original exception specification. Compilers typically offered no help in maintaining consistency among function implementations, exception specifications, and client code. Most programmers ultimately decided that C++98 exception specifications weren't worth the trouble.

During work on C++11, a consensus emerged that the truly meaningful information about a function's exception-emitting behavior was whether it had any. Black or white, either a function might emit an exception or it guaranteed that it wouldn't. This maybe-or-never dichotomy forms the basis of C++11's exception specifications, which essentially replace C++98's. (C++98-style exception specifications remain valid, but they're deprecated.) In C++11, unconditional `noexcept` is for functions that guarantee they won't emit exceptions.

Whether a function should be so declared is a matter of interface design. The exception-emitting behavior of a function is of key interest to clients. Callers can query a function's `noexcept` status, and the results of such a query can affect the exception safety or efficiency of the calling code. As such, whether a function is `noexcept` is as important a piece of information as whether a member function is `const`. Failure to declare a function `noexcept` when you know that it won't emit an exception is simply poor interface specification.

But there's an additional incentive to apply `noexcept` to functions that won't produce exceptions: it permits compilers to generate better object code. To understand why, it helps to examine the difference between the C++98 and C++11 ways of saying that a function won't emit exceptions. Consider a function `f` that promises callers they'll never receive an exception. The two ways of expressing that are:

```
int f(int x) throw();     // no exceptions from f: C++98 style

int f(int x) noexcept;    // no exceptions from f: C++11 style
```

If, at runtime, an exception leaves `f`, `f`'s exception specification is violated. With the C++98 exception specification, the call stack is unwound to `f`'s caller, and, after some actions not relevant here, program execution is terminated. With the C++11 exception specification, runtime behavior is slightly different: the stack is only *possibly* unwound before program execution is terminated.

The difference between unwinding the call stack and *possibly* unwinding it has a surprisingly large impact on code generation. In a `noexcept` function, optimizers need not keep the runtime stack in an unwindable state if an exception would propagate out of the function, nor must they ensure that objects in a `noexcept` function are destroyed in the inverse order of construction should an exception leave the function. Functions with "`throw()`" exception specifications lack such optimization flexibility, as do functions with no exception specification at all. The situation can be summarized this way:

```
RetType function(params) noexcept;     // most optimizable

RetType function(params) throw();      // less optimizable

RetType function(params);              // less optimizable
```

This alone is sufficient reason to declare functions `noexcept` whenever you know they won't produce exceptions.

For some functions, the case is even stronger. The move operations are the preeminent example. Suppose you have a C++98 code base making use of a `std::vector<Widget>`. `Widget`s are added to the `std::vector` from time to time via `push_back`:

```
std::vector<Widget> vw;

…

Widget w;

…                            // work with w

vw.push_back(w);             // add w to vw

…
```

Assume this code works fine, and you have no interest in modifying it for C++11. However, you do want to take advantage of the fact that C++11's move semantics can improve the performance of legacy code when move-enabled types are involved. You therefore ensure that `Widget` has move operations, either by writing them yourself or by seeing to it that the conditions for their automatic generation are fulfilled (see Item 17).

When a new element is added to a `std::vector`, it's possible that the `std::vector` lacks space for it, i.e., that the `std::vector`'s size is equal to its capacity. When that happens, the `std::vector` allocates a new, larger, chunk of memory to hold its elements, and it transfers the elements from the existing chunk of memory to the new one. In C++98, the transfer was accomplished by copying each element from the old memory to the new memory, then destroying the objects in the old memory. This approach enabled `push_back` to offer the strong exception safety guarantee: if an exception was thrown during the copying of the elements, the state of the `std::vector` remained unchanged, because none of the elements in the old memory were destroyed until all elements had been successfully copied into the new memory.

In C++11, a natural optimization would be to replace the copying of `std::vector` elements with moves. Unfortunately, doing this runs the risk of violating `push_back`'s exception safety guarantee. If *n* elements have been moved from the old memory and an exception is thrown moving element *n+1*, the `push_back` operation can't run to completion. But the original `std::vector` has been modified: *n* of its elements have been moved from. Restoring their original state may not be possible, because attempting to move each object back into the original memory may itself yield an exception.

This is a serious problem, because the behavior of legacy code could depend on `push_back`'s strong exception safety guarantee. Therefore, C++11 implementations can't silently replace copy operations inside `push_back` with moves unless it's known that the move operations won't emit exceptions. In that case, having moves replace copies would be safe, and the only side effect would be improved performance.

`std::vector::push_back` takes advantage of this "move if you can, but copy if you must" strategy, and it's not the only function in the Standard Library that does. Other functions sporting the strong exception safety guarantee in C++98 (e.g., `std::vector::reserve`, `std::deque::insert`, etc.) behave the same way. All these functions replace calls to copy operations in C++98 with calls to move operations in C++11 only if the move operations are known to not emit exceptions. But how can a function know if a move operation won't produce an exception? The answer is obvious: it checks to see if the operation is declared `noexcept`.[3]

`swap` functions comprise another case where `noexcept` is particularly desirable. `swap` is a key component of many STL algorithm implementations, and it's commonly employed in copy assignment operators, too. Its widespread use renders the optimizations that `noexcept` affords especially worthwhile. Interestingly, whether `swap`s in the Standard Library are `noexcept` is sometimes dependent on whether user-

3 The checking is typically rather roundabout. Functions like `std::vector::push_back` call `std::move_if_noexcept`, a variation of `std::move` that conditionally casts to an rvalue (see Item 23), depending on whether the type's move constructor is `noexcept`. In turn, `std::move_if_noexcept` consults `std::is_nothrow_move_constructible`, and the value of this type trait (see Item 9) is set by compilers, based on whether the move constructor has a `noexcept` (or `throw()`) designation.

defined swaps are noexcept. For example, the declarations for the Standard Library's swaps for arrays and std::pair are:

```
template <class T, size_t N>
void swap(T (&a)[N],                                    // see
          T (&b)[N]) noexcept(noexcept(swap(*a, *b)));  // below

template <class T1, class T2>
struct pair {
  …
  void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                              noexcept(swap(second, p.second)));
  …
};
```

These functions are *conditionally noexcept*: whether they are noexcept depends on whether the expressions inside the noexcept clauses are noexcept. Given two arrays of Widget, for example, swapping them is noexcept only if swapping individual elements in the arrays is noexcept, i.e., if swap for Widget is noexcept. The author of Widget's swap thus determines whether swapping arrays of Widget is noexcept. That, in turn, determines whether other swaps, such as the one for arrays of arrays of Widget, are noexcept. Similarly, whether swapping two std::pair objects containing Widgets is noexcept depends on whether swap for Widgets is noexcept. The fact that swapping higher-level data structures can generally be noexcept only if swapping their lower-level constituents is noexcept should motivate you to offer noexcept swap functions whenever you can.

By now, I hope you're excited about the optimization opportunities that noexcept affords. Alas, I must temper your enthusiasm. Optimization is important, but correctness is more important. I noted at the beginning of this Item that noexcept is part of a function's interface, so you should declare a function noexcept only if you are willing to commit to a noexcept implementation over the long term. If you declare a function noexcept and later regret that decision, your options are bleak. You can remove noexcept from the function's declaration (i.e., change its interface), thus running the risk of breaking client code. You can change the implementation such that an exception could escape, yet keep the original (now incorrect) exception specification. If you do that, your program will be terminated if an exception tries to leave the function. Or you can resign yourself to your existing implementation, abandoning whatever kindled your desire to change the implementation in the first place. None of these options is appealing.

The fact of the matter is that most functions are *exception-neutral*. Such functions throw no exceptions themselves, but functions they call might emit one. When that happens, the exception-neutral function allows the emitted exception to pass through on its way to a handler further up the call chain. Exception-neutral functions are never `noexcept`, because they may emit such "just passing through" exceptions. Most functions, therefore, quite properly lack the `noexcept` designation.

Some functions, however, have natural implementations that emit no exceptions, and for a few more—notably the move operations and `swap`—being `noexcept` can have such a significant payoff, it's worth implementing them in a `noexcept` manner if at all possible.[4] When you can honestly say that a function should never emit exceptions, you should definitely declare it `noexcept`.

Please note that I said some functions have *natural* `noexcept` implementations. Twisting a function's implementation to permit a `noexcept` declaration is the tail wagging the dog. Is putting the cart before the horse. Is not seeing the forest for the trees. Is…choose your favorite metaphor. If a straightforward function implementation might yield exceptions (e.g., by invoking a function that might throw), the hoops you'll jump through to hide that from callers (e.g., catching all exceptions and replacing them with status codes or special return values) will not only complicate your function's implementation, it will typically complicate code at call sites, too. For example, callers may have to check for status codes or special return values. The runtime cost of those complications (e.g., extra branches, larger functions that put more pressure on instruction caches, etc.) could exceed any speedup you'd hope to achieve via `noexcept`, plus you'd be saddled with source code that's more difficult to comprehend and maintain. That'd be poor software engineering.

For some functions, being `noexcept` is so important, they're that way by default. In C++98, it was considered bad style to permit the memory deallocation functions (i.e., `operator delete` and `operator delete[]`) and destructors to emit exceptions, and in C++11, this style rule has been all but upgraded to a language rule. By default, all memory deallocation functions and all destructors—both user-defined and compiler-generated—are implicitly `noexcept`. There's thus no need to declare them `noexcept`. (Doing so doesn't hurt anything, it's just unconventional.) The only time a destructor is not implicitly `noexcept` is when a data member of the class (including inherited members and those contained inside other data members) is of a type that expressly states that its destructor may emit exceptions (e.g., declares it "`noexcept(false)`"). Such destructors are uncommon. There are none in the Standard Library, and if the

4 The interface specifications for move operations on containers in the Standard Library lack `noexcept`. However, implementers are permitted to strengthen exception specifications for Standard Library functions, and, in practice, it is common for at least some container move operations to be declared `noexcept`. That practice exemplifies this Item's advice. Having found that it's possible to write container move operations such that exceptions aren't thrown, implementers often declare the operations `noexcept`, even though the Standard does not require them to do so.

destructor for an object being used by the Standard Library (e.g., because it's in a container or was passed to an algorithm) emits an exception, the behavior of the program is undefined.

It's worth noting that some library interface designers distinguish functions with *wide contracts* from those with *narrow contracts*. A function with a wide contract has no preconditions. Such a function may be called regardless of the state of the program, and it imposes no constraints on the arguments that callers pass it.[5] Functions with wide contracts never exhibit undefined behavior.

Functions without wide contracts have narrow contracts. For such functions, if a precondition is violated, results are undefined.

If you're writing a function with a wide contract and you know it won't emit exceptions, following the advice of this Item and declaring it `noexcept` is easy. For functions with narrow contracts, the situation is trickier. For example, suppose you're writing a function `f` taking a `std::string` parameter, and suppose `f`'s natural implementation never yields an exception. That suggests that `f` should be declared `noexcept`.

Now suppose that `f` has a precondition: the length of its `std::string` parameter doesn't exceed 32 characters. If `f` were to be called with a `std::string` whose length is greater than 32, behavior would be undefined, because a precondition violation *by definition* results in undefined behavior. `f` is under no obligation to check this precondition, because functions may assume that their preconditions are satisfied. (Callers are responsible for ensuring that such assumptions are valid.) Even with a precondition, then, declaring `f` `noexcept` seems appropriate:

```
void f(const std::string& s) noexcept;    // precondition:
                                          // s.length() <= 32
```

But suppose that `f`'s implementer chooses to check for precondition violations. Checking isn't required, but it's also not forbidden, and checking the precondition could be useful, e.g., during system testing. Debugging an exception that's been thrown is generally easier than trying to track down the cause of undefined behavior. But how should a precondition violation be reported such that a test harness or a client error handler could detect it? A straightforward approach would be to throw a "precondition was violated" exception, but if `f` is declared `noexcept`, that would be impossible; throwing an exception would lead to program termination. For this rea-

5 "Regardless of the state of the program" and "no constraints" doesn't legitimize programs whose behavior is already undefined. For example, `std::vector::size` has a wide contract, but that doesn't require that it behave reasonably if you invoke it on a random chunk of memory that you've cast to a `std::vector`. The result of the cast is undefined, so there are no behavioral guarantees for the program containing the cast.

son, library designers who distinguish wide from narrow contracts generally reserve `noexcept` for functions with wide contracts.

As a final point, let me elaborate on my earlier observation that compilers typically offer no help in identifying inconsistencies between function implementations and their exception specifications. Consider this code, which is perfectly legal:

```
void setup();              // functions defined elsewhere
void cleanup();

void doWork() noexcept
{
  setup();                 // set up work to be done

  …                        // do the actual work

  cleanup();               // perform cleanup actions
}
```

Here, `doWork` is declared `noexcept`, even though it calls the non-`noexcept` functions `setup` and `cleanup`. This seems contradictory, but it could be that `setup` and `cleanup` document that they never emit exceptions, even though they're not declared that way. There could be good reasons for their non-`noexcept` declarations. For example, they might be part of a library written in C. (Even functions from the C Standard Library that have been moved into the `std` namespace lack exception specifications, e.g., `std::strlen` isn't declared `noexcept`.) Or they could be part of a C++98 library that decided not to use C++98 exception specifications and hasn't yet been revised for C++11.

Because there are legitimate reasons for `noexcept` functions to rely on code lacking the `noexcept` guarantee, C++ permits such code, and compilers generally don't issue warnings about it.

Things to Remember

- `noexcept` is part of a function's interface, and that means that callers may depend on it.
- `noexcept` functions are more optimizable than non-`noexcept` functions.
- `noexcept` is particularly valuable for the move operations, `swap`, memory deallocation functions, and destructors.
- Most functions are exception-neutral rather than `noexcept`.

Item 15: Use `constexpr` whenever possible.

If there were an award for the most confusing new word in C++11, `constexpr` would probably win it. When applied to objects, it's essentially a beefed-up form of `const`, but when applied to functions, it has a quite different meaning. Cutting through the confusion is worth the trouble, because when `constexpr` corresponds to what you want to express, you definitely want to use it.

Conceptually, `constexpr` indicates a value that's not only constant, it's known during compilation. The concept is only part of the story, though, because when `constexpr` is applied to functions, things are more nuanced than this suggests. Lest I ruin the surprise ending, for now I'll just say that you can't assume that the results of `constexpr` functions are `const`, nor can you take for granted that their values are known during compilation. Perhaps most intriguingly, these things are *features*. It's *good* that `constexpr` functions need not produce results that are `const` or known during compilation!

But let's begin with `constexpr` objects. Such objects are, in fact, `const`, and they do, in fact, have values that are known at compile time. (Technically, their values are determined during *translation*, and translation consists not just of compilation but also of linking. Unless you write compilers or linkers for C++, however, this has no effect on you, so you can blithely program as if the values of `constexpr` objects were determined during compilation.)

Values known during compilation are privileged. They may be placed in read-only memory, for example, and, especially for developers of embedded systems, this can be a feature of considerable importance. Of broader applicability is that integral values that are constant and known during compilation can be used in contexts where C++ requires an *integral constant expression*. Such contexts include specification of array sizes, integral template arguments (including lengths of `std::array` objects), enumerator values, alignment specifiers, and more. If you want to use a variable for these kinds of things, you certainly want to declare it `constexpr`, because then compilers will ensure that it has a compile-time value:

```
int sz;                             // non-constexpr variable

…

constexpr auto arraySize1 = sz;     // error! sz's value not
                                    // known at compilation

std::array<int, sz> data1;          // error! same problem

constexpr auto arraySize2 = 10;     // fine, 10 is a
```

```
                                  // compile-time constant

std::array<int, arraySize2> data2;  // fine, arraySize2
                                    // is constexpr
```

Note that `const` doesn't offer the same guarantee as `constexpr`, because `const` objects need not be initialized with values known during compilation:

```
int sz;                             // as before

…

const auto arraySize = sz;          // fine, arraySize is
                                    // const copy of sz

std::array<int, arraySize> data;    // error! arraySize's value
                                    // not known at compilation
```

Simply put, all `constexpr` objects are `const`, but not all `const` objects are `constexpr`. If you want compilers to guarantee that a variable has a value that can be used in contexts requiring compile-time constants, the tool to reach for is `constexpr`, not `const`.

Usage scenarios for `constexpr` objects become more interesting when `constexpr` functions are involved. Such functions produce compile-time constants *when they are called with compile-time constants*. If they're called with values not known until runtime, they produce runtime values. This may sound as if you don't know what they'll do, but that's the wrong way to think about it. The right way to view it is this:

- `constexpr` functions can be used in contexts that demand compile-time constants. If the values of the arguments you pass to a `constexpr` function in such a context are known during compilation, the result will be computed during compilation. If any of the arguments' values is not known during compilation, your code will be rejected.
- When a `constexpr` function is called with one or more values that are not known during compilation, it acts like a normal function, computing its result at runtime. This means you don't need two functions to perform the same operation, one for compile-time constants and one for all other values. The `constexpr` function does it all.

Suppose we need a data structure to hold the results of an experiment that can be run in a variety of ways. For example, the lighting level can be high, low, or off during the course of the experiment, as can the fan speed and the temperature, etc. If there are *n* environmental conditions relevant to the experiment, each of which has three possible states, the number of combinations is 3^n. Storing experimental results for all combinations of conditions thus requires a data structure with enough room for 3^n values. Assuming each result is an `int` and that *n* is known (or can be computed) during compilation, a `std::array` could be a reasonable data structure choice. But we'd need a way to compute 3^n during compilation. The C++ Standard Library provides `std::pow`, which is the mathematical functionality we need, but, for our purposes, there are two problems with it. First, `std::pow` works on floating-point types, and we need an integral result. Second, `std::pow` isn't `constexpr` (i.e., isn't guaranteed to return a compile-time result when called with compile-time values), so we can't use it to specify a `std::array`'s size.

Fortunately, we can write the `pow` we need. I'll show how to do that in a moment, but first let's look at how it could be declared and used:

```
constexpr                              // pow's a constexpr func
int pow(int base, int exp) noexcept    // that never throws
{
  …                                    // impl is below
}

constexpr auto numConds = 5;                 // # of conditions

std::array<int, pow(3, numConds)> results;   // results has
                                             // 3^numConds
                                             // elements
```

Recall that the `constexpr` in front of `pow` doesn't say that `pow` returns a `const` value, it says that if `base` and `exp` are compile-time constants, `pow`'s result may be used as a compile-time constant. If `base` and/or `exp` are not compile-time constants, `pow`'s result will be computed at runtime. That means that `pow` can not only be called to do things like compile-time-compute the size of a `std::array`, it can also be called in runtime contexts such as this:

```
auto base = readFromDB("base");        // get these values
auto exp = readFromDB("exponent");     // at runtime

auto baseToExp = pow(base, exp);       // call pow function
                                       // at runtime
```

Because `constexpr` functions must be able to return compile-time results when called with compile-time values, restrictions are imposed on their implementations. The restrictions differ between C++11 and C++14.

In C++11, `constexpr` functions may contain no more than a single executable statement: a `return`. That sounds more limiting than it is, because two tricks can be used to extend the expressiveness of `constexpr` functions beyond what you might think. First, the conditional "`?:`" operator can be used in place of `if-else` statements, and second, recursion can be used instead of loops. `pow` can therefore be implemented like this:

```
constexpr int pow(int base, int exp) noexcept
{
  return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

This works, but it's hard to imagine that anybody except a hard-core functional programmer would consider it pretty. In C++14, the restrictions on `constexpr` functions are substantially looser, so the following implementation becomes possible:

```
constexpr int pow(int base, int exp) noexcept       // C++14
{
  auto result = 1;
  for (int i = 0; i < exp; ++i) result *= base;

  return result;
}
```

`constexpr` functions are limited to taking and returning *literal types*, which essentially means types that can have values determined during compilation. In C++11, all built-in types except `void` qualify, but user-defined types may be literal, too, because constructors and other member functions may be `constexpr`:

```
class Point {
public:
  constexpr Point(double xVal = 0, double yVal = 0) noexcept
  : x(xVal), y(yVal)
  {}

  constexpr double xValue() const noexcept { return x; }
  constexpr double yValue() const noexcept { return y; }

  void setX(double newX) noexcept { x = newX; }
  void setY(double newY) noexcept { y = newY; }

private:
  double x, y;
};
```

Here, the `Point` constructor can be declared `constexpr`, because if the arguments passed to it are known during compilation, the value of the data members of the constructed `Point` can also be known during compilation. `Point`s so initialized could thus be `constexpr`:

```
constexpr Point p1(9.4, 27.7);      // fine, "runs" constexpr
                                    // ctor during compilation

constexpr Point p2(28.8, 5.3);      // also fine
```

Similarly, the getters `xValue` and `yValue` can be `constexpr`, because if they're invoked on a `Point` object with a value known during compilation (e.g., a `constexpr Point` object), the values of the data members `x` and `y` can be known during compilation. That makes it possible to write `constexpr` functions that call `Point`'s getters and to initialize `constexpr` objects with the results of such functions:

```
constexpr
Point midpoint(const Point& p1, const Point& p2) noexcept
{
  return { (p1.xValue() + p2.xValue()) / 2,    // call constexpr
           (p1.yValue() + p2.yValue()) / 2 };  // member funcs
}

constexpr auto mid = midpoint(p1, p2);     // init constexpr
                                           // object w/result of
                                           // constexpr function
```

This is very exciting. It means that the object `mid`, though its initialization involves calls to constructors, getters, and a non-member function, can be created in read-only memory! It means you could use an expression like `mid.xValue() * 10` in an argument to a template or in an expression specifying the value of an enumerator![6] It means that the traditionally fairly strict line between work done during compilation and work done at runtime begins to blur, and some computations traditionally done at runtime can migrate to compile time. The more code taking part in the migration, the faster your software will run. (Compilation may take longer, however.)

In C++11, two restrictions prevent `Point`'s member functions `setX` and `setY` from being declared `constexpr`. First, they modify the object they operate on, and in C++11, `constexpr` member functions are implicitly `const`. Second, they have `void` return types, and `void` isn't a literal type in C++11. Both these restrictions are lifted in C++14, so in C++14, even `Point`'s setters can be `constexpr`:

6 Because `Point::xValue` returns `double`, the type of `mid.xValue() * 10` is also `double`. Floating-point types can't be used to instantiate templates or to specify enumerator values, but they can be used as part of larger expressions that yield integral types. For example, `static_cast<int>(mid.xValue() * 10)` could be used to instantiate a template or to specify an enumerator value.

```
class Point {
public:
  …

  constexpr void setX(double newX) noexcept     // C++14
  { x = newX; }

  constexpr void setY(double newY) noexcept     // C++14
  { y = newY; }

  …

};
```

That makes it possible to write functions like this:

```
// return reflection of p with respect to the origin (C++14)
constexpr Point reflection(const Point& p) noexcept
{
  Point result;                       // create non-const Point

  result.setX(-p.xValue());           // set its x and y values
  result.setY(-p.yValue());

  return result;                      // return copy of it
}
```

Client code could look like this:

```
constexpr Point p1(9.4, 27.7);        // as above
constexpr Point p2(28.8, 5.3);
constexpr auto mid = midpoint(p1, p2);

constexpr auto reflectedMid =         // reflectedMid's value is
  reflection(mid);                    // (-19.1 -16.5) and known
                                      // during compilation
```

The advice of this Item is to use `constexpr` whenever possible, and by now I hope it's clear why: both `constexpr` objects and `constexpr` functions can be employed in a wider range of contexts than non-`constexpr` objects and functions. By using `constexpr` whenever possible, you maximize the range of situations in which your objects and functions may be used.

It's important to note that `constexpr` is part of an object's or function's interface. `constexpr` proclaims "I can be used in a context where C++ requires a constant expression." If you declare an object or function `constexpr`, clients may use it in such contexts. If you later decide that your use of `constexpr` was a mistake and you remove it, you may cause arbitrarily large amounts of client code to stop compiling. (The simple act of adding I/O to a function for debugging or performance tuning could lead to such a problem, because I/O statements are generally not permitted in `constexpr` functions.) Part of "whenever possible" in "Use `constexpr` whenever possible" is your willingness to make a long-term commitment to the constraints it imposes on the objects and functions you apply it to.

Things to Remember

- `constexpr` objects are `const` and are initialized with values known during compilation.
- `constexpr` functions can produce compile-time results when called with arguments whose values are known during compilation.
- `constexpr` objects and functions may be used in a wider range of contexts than non-`constexpr` objects and functions.
- `constexpr` is part of an object's or function's interface.

Item 16: Make `const` member functions thread safe.

If we're working in a mathematical domain, we might find it convenient to have a class representing polynomials. Within this class, it would probably be useful to have a function to compute the root(s) of a polynomial, i.e., values where the polynomial evaluates to zero. Such a function would not modify the polynomial, so it'd be natural to declare it `const`:

```
class Polynomial {
public:
  using RootsType =           // data structure holding values
    std::vector<double>;      // where polynomial evals to zero
  …                           // (see Item 9 for info on "using")

  RootsType roots() const;

  …

};
```

Computing the roots of a polynomial can be expensive, so we don't want to do it if we don't have to. And if we do have to do it, we certainly don't want to do it more than once. We'll thus cache the root(s) of the polynomial if we have to compute them, and we'll implement `roots` to return the cached value. Here's the basic approach:

```
class Polynomial {
public:
  using RootsType = std::vector<double>;

  RootsType roots() const
  {
    if (!rootsAreValid) {            // if cache not valid

      …                              // compute roots,
                                     // store them in rootVals

      rootsAreValid = true;
    }

    return rootVals;
  }

private:
  mutable bool rootsAreValid{ false };    // see Item 7 for info
  mutable RootsType rootVals{};           // on initializers
};
```

Conceptually, `roots` doesn't change the `Polynomial` object on which it operates, but, as part of its caching activity, it may need to modify `rootVals` and `rootsAreValid`. That's a classic use case for `mutable`, and that's why it's part of the declarations for these data members.

Imagine now that two threads simultaneously call `roots` on a `Polynomial` object:

```
Polynomial p;

…

/*-----  Thread 1  ----- */      /*-------  Thread 2  ------- */

auto rootsOfP = p.roots();       auto valsGivingZero = p.roots();
```

This client code is perfectly reasonable. `roots` is a `const` member function, and that means it represents a read operation. Having multiple threads perform a read operation without synchronization is safe. At least it's supposed to be. In this case, it's not, because inside `roots`, one or both of these threads might try to modify the data members `rootsAreValid` and `rootVals`. That means that this code could have different threads reading and writing the same memory without synchronization, and that's the definition of a data race. This code has undefined behavior.

The problem is that `roots` is declared `const`, but it's not thread safe. The `const` declaration is as correct in C++11 as it would be in C++98 (retrieving the roots of a polynomial doesn't change the value of the polynomial), so what requires rectification is the lack of thread safety.

The easiest way to address the issue is the usual one: employ a `mutex`:

```
class Polynomial {
public:
  using RootsType = std::vector<double>;

  RootsType roots() const
  {
    std::lock_guard<std::mutex> g(m);     // lock mutex

    if (!rootsAreValid) {                 // if cache not valid

      …                                   // compute/store roots

      rootsAreValid = true;
    }

    return rootVals;
  }                                       // unlock mutex

private:
  mutable std::mutex m;
  mutable bool rootsAreValid{ false };
  mutable RootsType rootVals{};
};
```

The `std::mutex m` is declared `mutable`, because locking and unlocking it are non-`const` member functions, and within `roots` (a `const` member function), `m` would otherwise be considered a `const` object.

It's worth noting that because `std::mutex` is a *move-only type* (i.e., a type that can be moved, but not copied), a side effect of adding `m` to `Polynomial` is that `Polynomial` loses the ability to be copied. It can still be moved, however.

In some situations, a mutex is overkill. For example, if all you're doing is counting how many times a member function is called, a `std::atomic` counter (i.e, one where other threads are guaranteed to see its operations occur indivisibly—see Item 40) will often be a less expensive way to go. (Whether it actually is less expensive depends on the hardware you're running on and the implementation of mutexes in your Standard Library.) Here's how you can employ a `std::atomic` to count calls:

```
class Point {                                   // 2D point
public:
  …

  double distanceFromOrigin() const noexcept    // see Item 14
  {                                             // for noexcept

    ++callCount;                                // atomic increment

    return std::sqrt((x * x) + (y * y));
  }

private:
  mutable std::atomic<unsigned> callCount{ 0 };
  double x, y;
};
```

Like `std::mutex`es, `std::atomic`s are move-only types, so the existence of `callCount` in `Point` means that `Point` is also move-only.

Because operations on `std::atomic` variables are often less expensive than mutex acquisition and release, you may be tempted to lean on `std::atomic`s more heavily than you should. For example, in a class caching an expensive-to-compute `int`, you might try to use a pair of `std::atomic` variables instead of a mutex:

```
class Widget {
public:
  …

  int magicValue() const
  {
    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cachedValue = val1 + val2;              // uh oh, part 1
      cacheValid = true;                      // uh oh, part 2
      return cachedValue;
    }
  }

private:
```

```
  mutable std::atomic<bool> cacheValid{ false };
  mutable std::atomic<int> cachedValue;
};
```

This will work, but sometimes it will work a lot harder than it should. Consider:

- A thread calls `Widget::magicValue`, sees `cacheValid` as `false`, performs the two expensive computations, and assigns their sum to `cachedValue`.
- At that point, a second thread calls `Widget::magicValue`, also sees `cacheValid` as `false`, and thus carries out the same expensive computations that the first thread has just finished. (This "second thread" may in fact be *several* other threads.)

Such behavior is contrary to the goal of caching. Reversing the order of the assignments to `cachedValue` and `CacheValid` eliminates that problem, but the result is even worse:

```
class Widget {
public:
  …

  int magicValue() const
  {
    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cacheValid = true;                        // uh oh, part 1
      return cachedValue = val1 + val2;         // uh oh, part 2
    }
  }

  …

};
```

Imagine that `cacheValid` is `false`, and then:

- One thread calls `Widget::magicValue` and executes through the point where `cacheValid` is set to `true`.
- At that moment, a second thread calls `Widget::magicValue` and checks `cache Valid`. Seeing it `true`, the thread returns `cachedValue`, even though the first

thread has not yet made an assignment to it. The returned value is therefore incorrect.

There's a lesson here. For a single variable or memory location requiring synchronization, use of a `std::atomic` is adequate, but once you get to two or more variables or memory locations that require manipulation as a unit, you should reach for a mutex. For `Widget::magicValue`, that would look like this:

```
class Widget {
public:
  …

  int magicValue() const
  {
    std::lock_guard<std::mutex> guard(m);   // lock m

    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cachedValue = val1 + val2;
      cacheValid = true;
      return cachedValue;
    }
  }                                         // unlock m
  …

private:
  mutable std::mutex m;
  mutable int cachedValue;                  // no longer atomic
  mutable bool cacheValid{ false };         // no longer atomic
};
```

Now, this Item is predicated on the assumption that multiple threads may simultaneously execute a `const` member function on an object. If you're writing a `const` member function where that's not the case—where you can *guarantee* that there will never be more than one thread executing that member function on an object—the thread safety of the function is immaterial. For example, it's unimportant whether member functions of classes designed for exclusively single-threaded use are thread safe. In such cases, you can avoid the costs associated with mutexes and `std::atomic`s, as well as the side effect of their rendering the classes containing them move-only. However, such threading-free scenarios are increasingly uncommon, and they're likely to become rarer still. The safe bet is that `const` member functions will be subject to concurrent execution, and that's why you should ensure that your `const` member functions are thread safe.

Things to Remember

- Make `const` member functions thread safe unless you're *certain* they'll never be used in a concurrent context.
- Use of `std::atomic` variables may offer better performance than a mutex, but they're suited for manipulation of only a single variable or memory location.

Item 17: Understand special member function generation.

In official C++ parlance, the *special member functions* are the ones that C++ is willing to generate on its own. C++98 has four such functions: the default constructor, the destructor, the copy constructor, and the copy assignment operator. There's fine print, of course. These functions are generated only if they're needed, i.e., if some code uses them without their being expressly declared in the class. A default constructor is generated only if the class declares no constructors at all. (This prevents compilers from creating a default constructor for a class where you've specified that constructor arguments are required.) Generated special member functions are implicitly public and `inline`, and they're nonvirtual unless the function in question is a destructor in a derived class inheriting from a base class with a virtual destructor. In that case, the compiler-generated destructor for the derived class is also virtual.

But you already know these things. Yes, yes, ancient history: Mesopotamia, the Shang dynasty, FORTRAN, C++98. But times have changed, and the rules for special member function generation in C++ have changed with them. It's important to be aware of the new rules, because few things are as central to effective C++ programming as knowing when compilers silently insert member functions into your classes.

As of C++11, the special member functions club has two more inductees: the move constructor and the move assignment operator. Their signatures are:

```
class Widget {
public:
  …
  Widget(Widget&& rhs);              // move constructor

  Widget& operator=(Widget&& rhs);   // move assignment operator
  …
};
```

The rules governing their generation and behavior are analogous to those for their copying siblings. The move operations are generated only if they're needed, and if they are generated, they perform "memberwise moves" on the non-static data members of the class. That means that the move constructor move-constructs each non-static data member of the class from the corresponding member of its parameter `rhs`, and the move assignment operator move-assigns each non-static data member from its parameter. The move constructor also move-constructs its base class parts (if there are any), and the move assignment operator move-assigns its base class parts.

Now, when I refer to a move operation move-constructing or move-assigning a data member or base class, there is no guarantee that a move will actually take place. "Memberwise moves" are, in reality, more like memberwise move *requests*, because types that aren't *move-enabled* (i.e., that offer no special support for move operations, e.g., most C++98 legacy classes) will be "moved" via their copy operations. The heart of each memberwise "move" is application of `std::move` to the object to be moved from, and the result is used during function overload resolution to determine whether a move or a copy should be performed. Item 23 covers this process in detail. For this Item, simply remember that a memberwise move consists of move operations on data members and base classes that support move operations, but a copy operation for those that don't.

As is the case with the copy operations, the move operations aren't generated if you declare them yourself. However, the precise conditions under which they are generated differ a bit from those for the copy operations.

The two copy operations are independent: declaring one doesn't prevent compilers from generating the other. So if you declare a copy constructor, but no copy assignment operator, then write code that requires copy assignment, compilers will generate the copy assignment operator for you. Similarly, if you declare a copy assignment operator, but no copy constructor, yet your code requires copy construction, compilers will generate the copy constructor for you. That was true in C++98, and it's still true in C++11.

The two move operations are not independent. If you declare either, that prevents compilers from generating the other. The rationale is that if you declare, say, a move constructor for your class, you're indicating that there's something about how move construction should be implemented that's different from the default memberwise move that compilers would generate. And if there's something wrong with memberwise move construction, there'd probably be something wrong with memberwise move assignment, too. So declaring a move constructor prevents a move assignment operator from being generated, and declaring a move assignment operator prevents compilers from generating a move constructor.

Furthermore, move operations won't be generated for any class that explicitly declares a copy operation. The justification is that declaring a copy operation (construction or assignment) indicates that the normal approach to copying an object (memberwise copy) isn't appropriate for the class, and compilers figure that if memberwise copy isn't appropriate for the copy operations, memberwise move probably isn't appropriate for the move operations.

This goes in the other direction, too. Declaring a move operation (construction or assignment) in a class causes compilers to disable the copy operations. (The copy operations are disabled by deleting them—see Item 11). After all, if memberwise move isn't the proper way to move an object, there's no reason to expect that memberwise copy is the proper way to copy it. This may sound like it could break C++98 code, because the conditions under which the copy operations are enabled are more constrained in C++11 than in C++98, but this is not the case. C++98 code can't have move operations, because there was no such thing as "moving" objects in C++98. The only way a legacy class can have user-declared move operations is if they were added for C++11, and classes that are modified to take advantage of move semantics have to play by the C++11 rules for special member function generation.

Perhaps you've heard of a guideline known as the *Rule of Three*. The Rule of Three states that if you declare any of a copy constructor, copy assignment operator, or destructor, you should declare all three. It grew out of the observation that the need to take over the meaning of a copy operation almost always stemmed from the class performing some kind of resource management, and that almost always implied that (1) whatever resource management was being done in one copy operation probably needed to be done in the other copy operation and (2) the class destructor would also be participating in management of the resource (usually releasing it). The classic resource to be managed was memory, and this is why all Standard Library classes that manage memory (e.g., the STL containers that perform dynamic memory management) all declare "the big three": both copy operations and a destructor.

A consequence of the Rule of Three is that the presence of a user-declared destructor indicates that simple memberwise copy is unlikely to be appropriate for the copying operations in the class. That, in turn, suggests that if a class declares a destructor, the copy operations probably shouldn't be automatically generated, because they wouldn't do the right thing. At the time C++98 was adopted, the significance of this line of reasoning was not fully appreciated, so in C++98, the existence of a user-declared destructor had no impact on compilers' willingness to generate copy operations. That continues to be the case in C++11, but only because restricting the conditions under which the copy operations are generated would break too much legacy code.

The reasoning behind the Rule of Three remains valid, however, and that, combined with the observation that declaration of a copy operation precludes the implicit generation of the move operations, motivates the fact that C++11 does *not* generate move operations for a class with a user-declared destructor.

So move operations are generated for classes (when needed) only if these three things are true:

- No copy operations are declared in the class.
- No move operations are declared in the class.
- No destructor is declared in the class.

At some point, analogous rules may be extended to the copy operations, because C++11 deprecates the automatic generation of copy operations for classes declaring copy operations or a destructor. This means that if you have code that depends on the generation of copy operations in classes declaring a destructor or one of the copy operations, you should consider upgrading these classes to eliminate the dependence. Provided the behavior of the compiler-generated functions is correct (i.e, if memberwise copying of the class's non-static data members is what you want), your job is easy, because C++11's "`= default`" lets you say that explicitly:

```
class Widget {
public:
  …
  ~Widget();                              // user-declared dtor

  …                                       // default copy ctor
  Widget(const Widget&) = default;        // behavior is OK

  Widget&                                 // default copy assign
    operator=(const Widget&) = default;   // behavior is OK
  …
};
```

This approach is often useful in polymorphic base classes, i.e., classes defining interfaces through which derived class objects are manipulated. Polymorphic base classes normally have virtual destructors, because if they don't, some operations (e.g., the use of `delete` or `typeid` on a derived class object through a base class pointer or reference) yield undefined or misleading results. Unless a class inherits a destructor that's already virtual, the only way to make a destructor virtual is to explicitly declare it that way. Often, the default implementation would be correct, and "`= default`" is a good way to express that. However, a user-declared destructor suppresses generation of the move operations, so if movability is to be supported, "`= default`" often finds a second application. Declaring the move operations disables the copy operations, so if copyability is also desired, one more round of "`= default`" does the job:

```
class Base {
public:
  virtual ~Base() = default;                // make dtor virtual
```

```
  Base(Base&&) = default;                    // support moving
  Base& operator=(Base&&) = default;

  Base(const Base&) = default;               // support copying
  Base& operator=(const Base&) = default;

  …

};
```

In fact, even if you have a class where compilers are willing to generate the copy and move operations and where the generated functions would behave as you want, you may choose to adopt a policy of declaring them yourself and using "`= default`" for their definitions. It's more work, but it makes your intentions clearer, and it can help you sidestep some fairly subtle bugs. For example, suppose you have a class representing a string table, i.e., a data structure that permits fast lookups of string values via an integer ID:

```
class StringTable {
public:
  StringTable() {}
  …                  // functions for insertion, erasure, lookup,
                     // etc., but no copy/move/dtor functionality

private:
  std::map<int, std::string> values;
};
```

Assuming that the class declares no copy operations, no move operations, and no destructor, compilers will automatically generate these functions if they are used. That's very convenient.

But suppose that sometime later, it's decided that logging the default construction and the destruction of such objects would be useful. Adding that functionality is easy:

```
class StringTable {
public:
  StringTable()
  { makeLogEntry("Creating StringTable object"); }     // added

  ~StringTable()                                       // also
  { makeLogEntry("Destroying StringTable object"); }   // added

  …                                        // other funcs as before
```

```
  private:
    std::map<int, std::string> values;    // as before
  };
```

This looks reasonable, but declaring a destructor has a potentially significant side effect: it prevents the move operations from being generated. However, creation of the class's copy operations is unaffected. The code is therefore likely to compile, run, and pass its functional testing. That includes testing its move functionality, because even though this class is no longer move-enabled, requests to move it will compile and run. Such requests will, as noted earlier in this Item, cause copies to be made. Which means that code "moving" `StringTable` objects actually copies them, i.e., copies the underlying `std::map<int, std::string>` objects. And copying a `std::map<int, std::string>` is likely to be *orders of magnitude* slower than moving it. The simple act of adding a destructor to the class could thereby have introduced a significant performance problem! Had the copy and move operations been explicitly defined using "`= default`", the problem would not have arisen.

Now, having endured my endless blathering about the rules governing the copy and move operations in C++11, you may wonder when I'll turn my attention to the two other special member functions, the default constructor and the destructor. That time is now, but only for this sentence, because almost nothing has changed for these member functions: the rules in C++11 are nearly the same as in C++98.

The C++11 rules governing the special member functions are thus:

- **Default constructor**: Same rules as C++98. Generated only if the class contains no user-declared constructors.
- **Destructor**: Essentially same rules as C++98; sole difference is that destructors are `noexcept` by default (see Item 14). As in C++98, virtual only if a base class destructor is virtual.
- **Copy constructor**: Same runtime behavior as C++98: memberwise copy construction of non-static data members. Generated only if the class lacks a user-declared copy constructor. Deleted if the class declares a move operation. Generation of this function in a class with a user-declared copy assignment operator or destructor is deprecated.
- **Copy assignment operator**: Same runtime behavior as C++98: memberwise copy assignment of non-static data members. Generated only if the class lacks a user-declared copy assignment operator. Deleted if the class declares a move operation. Generation of this function in a class with a user-declared copy constructor or destructor is deprecated.

- **Move constructor** and **move assignment operator**: Each performs memberwise moving of non-static data members. Generated only if the class contains no user-declared copy operations, move operations, or destructor.

Note that there's nothing in the rules about the existence of a member function *template* preventing compilers from generating the special member functions. That means that if `Widget` looks like this,

```
class Widget {
  …
  template<typename T>                // construct Widget
  Widget(const T& rhs);               // from anything

  template<typename T>                // assign Widget
  Widget& operator=(const T& rhs);    // from anything
  …
};
```

compilers will still generate copy and move operations for `Widget` (assuming the usual conditions governing their generation are fulfilled), even though these templates could be instantiated to produce the signatures for the copy constructor and copy assignment operator. (That would be the case when `T` is `Widget`.) In all likelihood, this will strike you as an edge case barely worth acknowledging, but there's a reason I'm mentioning it. Item 26 demonstrates that it can have important consequences.

Things to Remember

- The special member functions are those compilers may generate on their own: default constructor, destructor, copy operations, and move operations.
- Move operations are generated only for classes lacking explicitly declared move operations, copy operations, and a destructor.
- The copy constructor is generated only for classes lacking an explicitly declared copy constructor, and it's deleted if a move operation is declared. The copy assignment operator is generated only for classes lacking an explicitly declared copy assignment operator, and it's deleted if a move operation is declared. Generation of the copy operations in classes with an explicitly declared destructor is deprecated.
- Member function templates never suppress generation of special member functions.

CHAPTER 4

Smart Pointers

Poets and songwriters have a thing about love. And sometimes about counting. Occasionally both. Inspired by the rather different takes on love and counting by Elizabeth Barrett Browning ("How do I love thee? Let me count the ways.") and Paul Simon ("There must be 50 ways to leave your lover."), we might try to enumerate the reasons why a raw pointer is hard to love:

1. Its declaration doesn't indicate whether it points to a single object or to an array.
2. Its declaration reveals nothing about whether you should destroy what it points to when you're done using it, i.e., if the pointer *owns* the thing it points to.
3. If you determine that you should destroy what the pointer points to, there's no way to tell how. Should you use `delete`, or is there a different destruction mechanism (e.g., a dedicated destruction function the pointer should be passed to)?
4. If you manage to find out that `delete` is the way to go, Reason 1 means it may not be possible to know whether to use the single-object form ("`delete`") or the array form ("`delete []`"). If you use the wrong form, results are undefined.
5. Assuming you ascertain that the pointer owns what it points to and you discover how to destroy it, it's difficult to ensure that you perform the destruction *exactly once* along every path in your code (including those due to exceptions). Missing a path leads to resource leaks, and doing the destruction more than once leads to undefined behavior.
6. There's typically no way to tell if the pointer dangles, i.e., points to memory that no longer holds the object the pointer is supposed to point to. Dangling pointers arise when objects are destroyed while pointers still point to them.

Raw pointers are powerful tools, to be sure, but decades of experience have demonstrated that with only the slightest lapse in concentration or discipline, these tools can turn on their ostensible masters.

Smart pointers are one way to address these issues. Smart pointers are wrappers around raw pointers that act much like the raw pointers they wrap, but that avoid many of their pitfalls. You should therefore prefer smart pointers to raw pointers. Smart pointers can do virtually everything raw pointers can, but with far fewer opportunities for error.

There are four smart pointers in C++11: `std::auto_ptr`, `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. All are designed to help manage the lifetimes of dynamically allocated objects, i.e., to avoid resource leaks by ensuring that such objects are destroyed in the appropriate manner at the appropriate time (including in the event of exceptions).

`std::auto_ptr` is a deprecated leftover from C++98. It was an attempt to standardize what later became C++11's `std::unique_ptr`. Doing the job right required move semantics, but C++98 didn't have them. As a workaround, `std::auto_ptr` co-opted its copy operations for moves. This led to surprising code (copying a `std::auto_ptr` sets it to null!) and frustrating usage restrictions (e.g., it's not possible to store `std::auto_ptr`s in containers).

`std::unique_ptr` does everything `std::auto_ptr` does, plus more. It does it as efficiently, and it does it without warping what it means to copy an object. It's better than `std::auto_ptr` in every way. The only legitimate use case for `std::auto_ptr` is a need to compile code with C++98 compilers. Unless you have that constraint, you should replace `std::auto_ptr` with `std::unique_ptr` and never look back.

The smart pointer APIs are remarkably varied. About the only functionality common to all is default construction. Because comprehensive references for these APIs are widely available, I'll focus my discussions on information that's often missing from API overviews, e.g., noteworthy use cases, runtime cost analyses, etc. Mastering such information can be the difference between merely using these smart pointers and using them *effectively*.

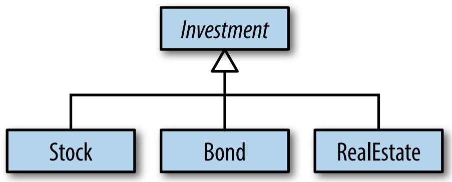
Item 18: Use `std::unique_ptr` for exclusive-ownership resource management.

When you reach for a smart pointer, `std::unique_ptr` should generally be the one closest at hand. It's reasonable to assume that, by default, `std::unique_ptr`s are the same size as raw pointers, and for most operations (including dereferencing), they execute exactly the same instructions. This means you can use them even in situations where memory and cycles are tight. If a raw pointer is small enough and fast enough for you, a `std::unique_ptr` almost certainly is, too.

`std::unique_ptr` embodies *exclusive ownership* semantics. A non-null `std::unique_ptr` always owns what it points to. Moving a `std::unique_ptr` transfers ownership from the source pointer to the destination pointer. (The source pointer is set to null.) Copying a `std::unique_ptr` isn't allowed, because if you could copy a `std::unique_ptr`, you'd end up with two `std::unique_ptr`s to the same resource, each thinking it owned (and should therefore destroy) that resource. `std::unique_ptr` is thus a *move-only type*. Upon destruction, a non-null `std::unique_ptr` destroys its resource. By default, resource destruction is accomplished by applying `delete` to the raw pointer inside the `std::unique_ptr`.

A common use for `std::unique_ptr` is as a factory function return type for objects in a hierarchy. Suppose we have a hierarchy for types of investments (e.g., stocks, bonds, real estate, etc.) with a base class `Investment`.

```
class Investment { … };

class Stock:
  public Investment { … };

class Bond:
  public Investment { … };

class RealEstate:
  public Investment { … };
```



A factory function for such a hierarchy typically allocates an object on the heap and returns a pointer to it, with the caller being responsible for deleting the object when it's no longer needed. That's a perfect match for `std::unique_ptr`, because the caller acquires responsibility for the resource returned by the factory (i.e., exclusive ownership of it), and the `std::unique_ptr` automatically deletes what it points to when the `std::unique_ptr` is destroyed. A factory function for the `Investment` hierarchy could be declared like this:

```
template<typename... Ts>           // return std::unique_ptr
std::unique_ptr<Investment>        // to an object created
makeInvestment(Ts&&... params);    // from the given args
```

Callers could use the returned `std::unique_ptr` in a single scope as follows,

```
{
  …
```

```
  auto pInvestment =                 // pInvestment is of type
    makeInvestment( arguments );     // std::unique_ptr<Investment>

  …

}                                    // destroy *pInvestment
```

but they could also use it in ownership-migration scenarios, such as when the `std::unique_ptr` returned from the factory is moved into a container, the container element is subsequently moved into a data member of an object, and that object is later destroyed. When that happens, the object's `std::unique_ptr` data member would also be destroyed, and its destruction would cause the resource returned from the factory to be destroyed. If the ownership chain got interrupted due to an exception or other atypical control flow (e.g., early function return or `break` from a loop), the `std::unique_ptr` owning the managed resource would eventually have its destructor called,[1] and the resource it was managing would thereby be destroyed.

By default, that destruction would take place via `delete`, but, during construction, `std::unique_ptr` objects can be configured to use *custom deleters*: arbitrary functions (or function objects, including those arising from lambda expressions) to be invoked when it's time for their resources to be destroyed. If the object created by `makeInvestment` shouldn't be directly `delete`d, but instead should first have a log entry written, `makeInvestment` could be implemented as follows. (An explanation follows the code, so don't worry if you see something whose motivation is less than obvious.)

```
auto delInvmt = [](Investment* pInvestment)       // custom
                {                                 // deleter
                  makeLogEntry(pInvestment);      // (a lambda
                  delete pInvestment;             // expression)
                };

template<typename... Ts>                          // revised
std::unique_ptr<Investment, decltype(delInvmt)>   // return type
makeInvestment(Ts&&... params)
{
  std::unique_ptr<Investment, decltype(delInvmt)> // ptr to be
    pInv(nullptr, delInvmt);                      // returned
```

1 There are a few exceptions to this rule. Most stem from abnormal program termination. If an exception propagates out of a thread's primary function (e.g., `main`, for the program's initial thread) or if a `noexcept` specification is violated (see Item 14), local objects may not be destroyed, and if `std::abort` or an exit function (i.e., `std::_Exit`, `std::exit`, or `std::quick_exit`) is called, they definitely won't be.

```
  if ( /* a Stock object should be created */ )
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( /* a Bond object should be created */ )
  {
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  }
  else if ( /* a RealEstate object should be created */ )
  {
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
  }

  return pInv;
}
```

In a moment, I'll explain how this works, but first consider how things look if you're a caller. Assuming you store the result of the `makeInvestment` call in an `auto` variable, you frolic in blissful ignorance of the fact that the resource you're using requires special treatment during deletion. In fact, you veritably bathe in bliss, because the use of `std::unique_ptr` means you need not concern yourself with when the resource should be destroyed, much less ensure that the destruction happens exactly once along every path through the program. `std::unique_ptr` takes care of all those things automatically. From a client's perspective, `makeInvestment`'s interface is sweet.

The implementation is pretty nice, too, once you understand the following:

- `delInvmt` is the custom deleter for the object returned from `makeInvestment`. All custom deletion functions accept a raw pointer to the object to be destroyed, then do what is necessary to destroy that object. In this case, the action is to call `makeLogEntry` and then apply `delete`. Using a lambda expression to create `delInvmt` is convenient, but, as we'll see shortly, it's also more efficient than writing a conventional function.
- When a custom deleter is to be used, its type must be specified as the second type argument to `std::unique_ptr`. In this case, that's the type of `delInvmt`, and that's why the return type of `makeInvestment` is `std::unique_ptr<Investment, decltype(delInvmt)>`. (For information about `decltype`, see Item 3.)
- The basic strategy of `makeInvestment` is to create a null `std::unique_ptr`, make it point to an object of the appropriate type, and then return it. To associate the custom deleter `delInvmt` with `pInv`, we pass that as its second constructor argument.

- Attempting to assign a raw pointer (e.g., from `new`) to a `std::unique_ptr` won't compile, because it would constitute an implicit conversion from a raw to a smart pointer. Such implicit conversions can be problematic, so C++11's smart pointers prohibit them. That's why `reset` is used to have `pInv` assume ownership of the object created via `new`.
- With each use of `new`, we use `std::forward` to perfect-forward the arguments passed to `makeInvestment` (see Item 25). This makes all the information provided by callers available to the constructors of the objects being created.
- The custom deleter takes a parameter of type `Investment*`. Regardless of the actual type of object created inside `makeInvestment` (i.e., `Stock`, `Bond`, or `RealEstate`), it will ultimately be `delete`d inside the lambda expression as an `Investment*` object. This means we'll be deleting a derived class object via a base class pointer. For that to work, the base class—`Investment`—must have a virtual destructor:

```
class Investment {
public:
  …                                            // essential
  virtual ~Investment();                       // design
  …                                            // component!
};
```

In C++14, the existence of function return type deduction (see Item 3) means that `makeInvestment` could be implemented in this simpler and more encapsulated fashion:

```
template<typename... Ts>
auto makeInvestment(Ts&&... params)                // C++14
{
  auto delInvmt = [](Investment* pInvestment)      // this is now
                  {                                // inside
                    makeLogEntry(pInvestment);     // make-
                    delete pInvestment;            // Investment
                  };

  std::unique_ptr<Investment, decltype(delInvmt)>   // as
    pInv(nullptr, delInvmt);                        // before

  if ( … )                                          // as before
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( … )                                     // as before
```

```
  {
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  }
  else if ( … )                                      // as before
  {
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
  }
  return pInv;                                       // as before
}
```

I remarked earlier that, when using the default deleter (i.e., `delete`), you can reasonably assume that `std::unique_ptr` objects are the same size as raw pointers. When custom deleters enter the picture, this may no longer be the case. Deleters that are function pointers generally cause the size of a `std::unique_ptr` to grow from one word to two. For deleters that are function objects, the change in size depends on how much state is stored in the function object. Stateless function objects (e.g., from lambda expressions with no captures) incur no size penalty, and this means that when a custom deleter can be implemented as either a function or a captureless lambda expression, the lambda is preferable:

```
auto delInvmt1 = [](Investment* pInvestment)       // custom
                 {                                 // deleter
                   makeLogEntry(pInvestment);      // as
                   delete pInvestment;             // stateless
                 };                                // lambda

template<typename... Ts>                           // return type
std::unique_ptr<Investment, decltype(delInvmt1)>   // has size of
makeInvestment(Ts&&... args);                      // Investment*

void delInvmt2(Investment* pInvestment)            // custom
{                                                  // deleter
  makeLogEntry(pInvestment);                       // as function
  delete pInvestment;
}

template<typename... Ts>                 // return type has
std::unique_ptr<Investment,              // size of Investment*
                void (*)(Investment*)>   // plus at least size
makeInvestment(Ts&&... params);          // of function pointer!
```

Function object deleters with extensive state can yield `std::unique_ptr` objects of significant size. If you find that a custom deleter makes your `std::unique_ptr`s unacceptably large, you probably need to change your design.

Factory functions are not the only common use case for std::unique_ptrs. They're even more popular as a mechanism for implementing the Pimpl Idiom. The code for that isn't complicated, but in some cases it's less than straightforward, so I'll refer you to Item 22, which is dedicated to the topic.

std::unique_ptr comes in two forms, one for individual objects (std::unique_ptr<T>) and one for arrays (std::unique_ptr<T[]>). As a result, there's never any ambiguity about what kind of entity a std::unique_ptr points to. The std::unique_ptr API is designed to match the form you're using. For example, there's no indexing operator (operator[]) for the single-object form, while the array form lacks dereferencing operators (operator* and operator->).

The existence of std::unique_ptr for arrays should be of only intellectual interest to you, because std::array, std::vector, and std::string are virtually always better data structure choices than raw arrays. About the only situation I can conceive of when a std::unique_ptr<T[]> would make sense would be when you're using a C-like API that returns a raw pointer to a heap array that you assume ownership of.

std::unique_ptr is the C++11 way to express exclusive ownership, but one of its most attractive features is that it easily and efficiently converts to a std::shared_ptr:

```
std::shared_ptr<Investment> sp =     // converts std::unique_ptr
  makeInvestment( arguments );       // to std::shared_ptr
```

This is a key part of why std::unique_ptr is so well suited as a factory function return type. Factory functions can't know whether callers will want to use exclusive-ownership semantics for the object they return or whether shared ownership (i.e., std::shared_ptr) would be more appropriate. By returning a std::unique_ptr, factories provide callers with the most efficient smart pointer, but they don't hinder callers from replacing it with its more flexible sibling. (For information about std::shared_ptr, proceed to Item 19.)

Things to Remember

- std::unique_ptr is a small, fast, move-only smart pointer for managing resources with exclusive-ownership semantics.
- By default, resource destruction takes place via delete, but custom deleters can be specified. Stateful deleters and function pointers as deleters increase the size of std::unique_ptr objects.
- Converting a std::unique_ptr to a std::shared_ptr is easy.

Item 19: Use `std::shared_ptr` for shared-ownership resource management.

Programmers using languages with garbage collection point and laugh at what C++ programmers go through to prevent resource leaks. "How primitive!" they jeer. "Didn't you get the memo from Lisp in the 1960s? Machines should manage resource lifetimes, not humans." C++ developers roll their eyes. "You mean the memo where the only resource is memory and the timing of resource reclamation is nondeterministic? We prefer the generality and predictability of destructors, thank you." But our bravado is part bluster. Garbage collection really is convenient, and manual lifetime management really can seem akin to constructing a mnemonic memory circuit using stone knives and bear skins. Why can't we have the best of both worlds: a system that works automatically (like garbage collection), yet applies to all resources and has predictable timing (like destructors)?

`std::shared_ptr` is the C++11 way of binding these worlds together. An object accessed via `std::shared_ptr`s has its lifetime managed by those pointers through *shared ownership*. No specific `std::shared_ptr` owns the object. Instead, all `std::shared_ptr`s pointing to it collaborate to ensure its destruction at the point where it's no longer needed. When the last `std::shared_ptr` pointing to an object stops pointing there (e.g., because the `std::shared_ptr` is destroyed or made to point to a different object), that `std::shared_ptr` destroys the object it points to. As with garbage collection, clients need not concern themselves with managing the lifetime of pointed-to objects, but as with destructors, the timing of the objects' destruction is deterministic.

A `std::shared_ptr` can tell whether it's the last one pointing to a resource by consulting the resource's *reference count*, a value associated with the resource that keeps track of how many `std::shared_ptr`s point to it. `std::shared_ptr` constructors increment this count (usually—see below), `std::shared_ptr` destructors decrement it, and copy assignment operators do both. (If `sp1` and `sp2` are `std::shared_ptr`s to different objects, the assignment "`sp1 = sp2;`" modifies `sp1` such that it points to the object pointed to by `sp2`. The net effect of the assignment is that the reference count for the object originally pointed to by `sp1` is decremented, while that for the object pointed to by `sp2` is incremented.) If a `std::shared_ptr` sees a reference count of zero after performing a decrement, no more `std::shared_ptr`s point to the resource, so the `std::shared_ptr` destroys it.

The existence of the reference count has performance implications:

- **`std::shared_ptr`s are twice the size of a raw pointer**, because they internally contain a raw pointer to the resource as well as a raw pointer to the resource's reference count.[2]
- **Memory for the reference count must be dynamically allocated**. Conceptually, the reference count is associated with the object being pointed to, but pointed-to objects know nothing about this. They thus have no place to store a reference count. (A pleasant implication is that any object—even those of built-in types—may be managed by `std::shared_ptr`s.) Item 21 explains that the cost of the dynamic allocation is avoided when the `std::shared_ptr` is created by `std::make_shared`, but there are situations where `std::make_shared` can't be used. Either way, the reference count is stored as dynamically allocated data.
- **Increments and decrements of the reference count must be atomic**, because there can be simultaneous readers and writers in different threads. For example, a `std::shared_ptr` pointing to a resource in one thread could be executing its destructor (hence decrementing the reference count for the resource it points to), while, in a different thread, a `std::shared_ptr` to the same object could be copied (and therefore incrementing the same reference count). Atomic operations are typically slower than non-atomic operations, so even though reference counts are usually only a word in size, you should assume that reading and writing them is comparatively costly.

Did I pique your curiosity when I wrote that `std::shared_ptr` constructors only "usually" increment the reference count for the object they point to? Creating a `std::shared_ptr` pointing to an object always yields one more `std::shared_ptr` pointing to that object, so why mustn't we *always* increment the reference count?

Move construction, that's why. Move-constructing a `std::shared_ptr` from another `std::shared_ptr` sets the source `std::shared_ptr` to null, and that means that the old `std::shared_ptr` stops pointing to the resource at the moment the new `std::shared_ptr` starts. As a result, no reference count manipulation is required. Moving `std::shared_ptr`s is therefore faster than copying them: copying requires incrementing the reference count, but moving doesn't. This is as true for assignment as for construction, so move construction is faster than copy construction, and move assignment is faster than copy assignment.

Like `std::unique_ptr` (see Item 18), `std::shared_ptr` uses `delete` as its default resource-destruction mechanism, but it also supports custom deleters. The design of this support differs from that for `std::unique_ptr`, however. For

2 This implementation is not required by the Standard, but every Standard Library implementation I'm familiar with employs it.

std::unique_ptr, the type of the deleter is part of the type of the smart pointer. For std::shared_ptr, it's not:

```
auto loggingDel = [](Widget *pw)        // custom deleter
                  {                     // (as in Item 18)
                    makeLogEntry(pw);
                    delete pw;
                  };

std::unique_ptr<                        // deleter type is
  Widget, decltype(loggingDel)          // part of ptr type
  > upw(new Widget, loggingDel);

std::shared_ptr<Widget>                 // deleter type is not
  spw(new Widget, loggingDel);          // part of ptr type
```

The std::shared_ptr design is more flexible. Consider two std::shared_ptr <Widget>s, each with a custom deleter of a different type (e.g., because the custom deleters are specified via lambda expressions):

```
auto customDeleter1 = [](Widget *pw) { … };    // custom deleters,
auto customDeleter2 = [](Widget *pw) { … };    // each with a
                                               // different type

std::shared_ptr<Widget> pw1(new Widget, customDeleter1);
std::shared_ptr<Widget> pw2(new Widget, customDeleter2);
```

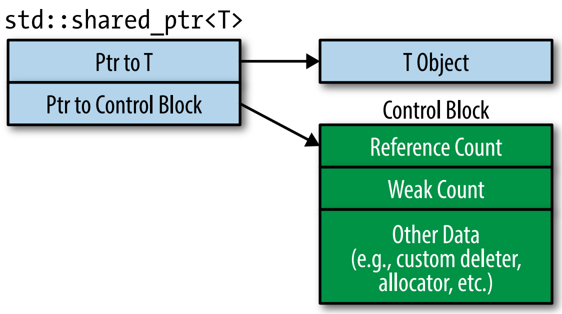
Because pw1 and pw2 have the same type, they can be placed in a container of objects of that type:

```
std::vector<std::shared_ptr<Widget>> vpw{ pw1, pw2 };
```

They could also be assigned to one another, and they could each be passed to a function taking a parameter of type std::shared_ptr<Widget>. None of these things can be done with std::unique_ptrs that differ in the types of their custom deleters, because the type of the custom deleter would affect the type of the std::unique_ptr.

In another difference from std::unique_ptr, specifying a custom deleter doesn't change the size of a std::shared_ptr object. Regardless of deleter, a std::shared_ptr object is two pointers in size. That's great news, but it should make you vaguely uneasy. Custom deleters can be function objects, and function objects can contain arbitrary amounts of data. That means they can be arbitrarily large. How can a std::shared_ptr refer to a deleter of arbitrary size without using any more memory?

It can't. It may have to use more memory. However, that memory isn't part of the `std::shared_ptr` object. It's on the heap or, if the creator of the `std::shared_ptr` took advantage of `std::shared_ptr` support for custom allocators, it's wherever the memory managed by the allocator is located. I remarked earlier that a `std::shared_ptr` object contains a pointer to the reference count for the object it points to. That's true, but it's a bit misleading, because the reference count is part of a larger data structure known as the *control block*. There's a control block for each object managed by `std::shared_ptr`s. The control block contains, in addition to the reference count, a copy of the custom deleter, if one has been specified. If a custom allocator was specified, the control block contains a copy of that, too. The control block may also contain additional data, including, as Item 21 explains, a secondary reference count known as the weak count, but we'll ignore such data in this Item. We can envision the memory associated with a `std::shared_ptr<T>` object as looking like this:



An object's control block is set up by the function creating the first `std::shared_ptr` to the object. At least that's what's supposed to happen. In general, it's impossible for a function creating a `std::shared_ptr` to an object to know whether some other `std::shared_ptr` already points to that object, so the following rules for control block creation are used:

- **`std::make_shared` (see Item 21) always creates a control block.** It manufactures a new object to point to, so there is certainly no control block for that object at the time `std::make_shared` is called.
- **A control block is created when a `std::shared_ptr` is constructed from a unique-ownership pointer (i.e., a `std::unique_ptr` or `std::auto_ptr`).** Unique-ownership pointers don't use control blocks, so there should be no control block for the pointed-to object. (As part of its construction, the `std::shared_ptr` assumes ownership of the pointed-to object, so the unique-ownership pointer is set to null.)

- **When a `std::shared_ptr` constructor is called with a raw pointer, it creates a control block.** If you wanted to create a `std::shared_ptr` from an object that already had a control block, you'd presumably pass a `std::shared_ptr` or a `std::weak_ptr` (see Item 20) as a constructor argument, not a raw pointer. `std::shared_ptr` constructors taking `std::shared_ptr`s or `std::weak_ptr`s as constructor arguments don't create new control blocks, because they can rely on the smart pointers passed to them to point to any necessary control blocks.

A consequence of these rules is that constructing more than one `std::shared_ptr` from a single raw pointer gives you a complimentary ride on the particle accelerator of undefined behavior, because the pointed-to object will have multiple control blocks. Multiple control blocks means multiple reference counts, and multiple reference counts means the object will be destroyed multiple times (once for each reference count). That means that code like this is bad, bad, bad:

```
auto pw = new Widget;                          // pw is raw ptr

…

std::shared_ptr<Widget> spw1(pw, loggingDel);  // create control
                                               // block for *pw
…

std::shared_ptr<Widget> spw2(pw, loggingDel);  // create 2nd
                                               // control block
                                               // for *pw!
```

The creation of the raw pointer `pw` to a dynamically allocated object is bad, because it runs contrary to the advice behind this entire chapter: to prefer smart pointers to raw pointers. (If you've forgotten the motivation for that advice, refresh your memory on page 115.) But set that aside. The line creating `pw` is a stylistic abomination, but at least it doesn't cause undefined program behavior.

Now, the constructor for `spw1` is called with a raw pointer, so it creates a control block (and thereby a reference count) for what's pointed to. In this case, that's `*pw` (i.e., the object pointed to by `pw`). In and of itself, that's okay, but the constructor for `spw2` is called with the same raw pointer, so it also creates a control block (hence a reference count) for `*pw`. `*pw` thus has two reference counts, each of which will eventually become zero, and that will ultimately lead to an attempt to destroy `*pw` twice. The second destruction is responsible for the undefined behavior.

There are at least two lessons regarding `std::shared_ptr` use here. First, try to avoid passing raw pointers to a `std::shared_ptr` constructor. The usual alternative is to use `std::make_shared` (see Item 21), but in the example above, we're using custom deleters, and that's not possible with `std::make_shared`. Second, if you must pass a raw pointer to a `std::shared_ptr` constructor, pass the result of `new` directly instead of going through a raw pointer variable. If the first part of the code above were rewritten like this,

```
std::shared_ptr<Widget> spw1(new Widget,    // direct use of new
                             loggingDel);
```

it'd be a lot less tempting to create a second `std::shared_ptr` from the same raw pointer. Instead, the author of the code creating `spw2` would naturally use `spw1` as an initialization argument (i.e., would call the `std::shared_ptr` copy constructor), and that would pose no problem whatsoever:

```
std::shared_ptr<Widget> spw2(spw1);     // spw2 uses same
                                        // control block as spw1
```

An especially surprising way that using raw pointer variables as `std::shared_ptr` constructor arguments can lead to multiple control blocks involves the `this` pointer. Suppose our program uses `std::shared_ptr`s to manage `Widget` objects, and we have a data structure that keeps track of `Widget`s that have been processed:

```
std::vector<std::shared_ptr<Widget>> processedWidgets;
```

Further suppose that `Widget` has a member function that does the processing:

```
class Widget {
public:
  …
  void process();
  …
};
```

Here's a reasonable-looking approach for `Widget::process`:

```
void Widget::process()
{
  …                                       // process the Widget

  processedWidgets.emplace_back(this);    // add it to list of
}                                         // processed Widgets;
                                          // this is wrong!
```

The comment about this being wrong says it all—or at least most of it. (The part that's wrong is the passing of `this`, not the use of `emplace_back`. If you're not familiar with `emplace_back`, see Item 42.) This code will compile, but it's passing a raw pointer (`this`) to a container of `std::shared_ptr`s. The `std::shared_ptr` thus constructed will create a new control block for the pointed-to `Widget` (`*this`). That doesn't sound harmful until you realize that if there are `std::shared_ptr`s outside the member function that already point to that `Widget`, it's game, set, and match for undefined behavior.

The `std::shared_ptr` API includes a facility for just this kind of situation. It has probably the oddest of all names in the Standard C++ Library: `std::enable_shared_from_this`. That's a template for a base class you inherit from if you want a class managed by `std::shared_ptr`s to be able to safely create a `std::shared_ptr` from a `this` pointer. In our example, `Widget` would inherit from `std::enable_shared_from_this` as follows:

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  …
  void process();
  …
};
```

As I said, `std::enable_shared_from_this` is a base class template. Its type parameter is always the name of the class being derived, so `Widget` inherits from `std::enable_shared_from_this<Widget>`. If the idea of a derived class inheriting from a base class templatized on the derived class makes your head hurt, try not to think about it. The code is completely legal, and the design pattern behind it is so well established, it has a standard name, albeit one that's almost as odd as `std::enable_shared_from_this`. The name is *The Curiously Recurring Template Pattern (CRTP)*. If you'd like to learn more about it, unleash your search engine, because here we need to get back to `std::enable_shared_from_this`.

`std::enable_shared_from_this` defines a member function that creates a `std::shared_ptr` to the current object, but it does it without duplicating control blocks. The member function is `shared_from_this`, and you use it in member functions whenever you want a `std::shared_ptr` that points to the same object as the `this` pointer. Here's a safe implementation of `Widget::process`:

```
void Widget::process()
{
  // as before, process the Widget
  …

  // add std::shared_ptr to current object to processedWidgets
  processedWidgets.emplace_back(shared_from_this());
}
```

Internally, `shared_from_this` looks up the control block for the current object, and it creates a new `std::shared_ptr` that refers to that control block. The design relies on the current object having an associated control block. For that to be the case, there must be an existing `std::shared_ptr` (e.g., one outside the member function calling `shared_from_this`) that points to the current object. If no such `std::shared_ptr` exists (i.e., if the current object has no associated control block), behavior is undefined, although `shared_from_this` typically throws an exception.

To prevent clients from calling member functions that invoke `shared_from_this` before a `std::shared_ptr` points to the object, classes inheriting from `std::enable_shared_from_this` often declare their constructors `private` and have clients create objects by calling factory functions that return `std::shared_ptr`s. `Widget`, for example, could look like this:

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  // factory function that perfect-forwards args
  // to a private ctor
  template<typename... Ts>
  static std::shared_ptr<Widget> create(Ts&&... params);

  …
  void process();             // as before
  …

private:
  …                           // ctors
};
```

By now, you may only dimly recall that our discussion of control blocks was motivated by a desire to understand the costs associated with `std::shared_ptr`s. Now that we understand how to avoid creating too many control blocks, let's return to the original topic.

A control block is typically only a few words in size, although custom deleters and allocators may make it larger. The usual control block implementation is more sophisticated than you might expect. It makes use of inheritance, and there's even a virtual function. (It's used to ensure that the pointed-to object is properly destroyed.) That means that using `std::shared_ptr`s also incurs the cost of the machinery for the virtual function used by the control block.

Having read about dynamically allocated control blocks, arbitrarily large deleters and allocators, virtual function machinery, and atomic reference count manipulations, your enthusiasm for `std::shared_ptr`s may have waned somewhat. That's fine.

They're not the best solution to every resource management problem. But for the functionality they provide, `std::shared_ptr`s exact a very reasonable cost. Under typical conditions, where the default deleter and default allocator are used and where the `std::shared_ptr` is created by `std::make_shared`, the control block is only about three words in size, and its allocation is essentially free. (It's incorporated into the memory allocation for the object being pointed to. For details, see Item 21.) Dereferencing a `std::shared_ptr` is no more expensive than dereferencing a raw pointer. Performing an operation requiring a reference count manipulation (e.g., copy construction or copy assignment, destruction) entails one or two atomic operations, but these operations typically map to individual machine instructions, so although they may be expensive compared to non-atomic instructions, they're still just single instructions. The virtual function machinery in the control block is generally used only once per object managed by `std::shared_ptr`s: when the object is destroyed.

In exchange for these rather modest costs, you get automatic lifetime management of dynamically allocated resources. Most of the time, using `std::shared_ptr` is vastly preferable to trying to manage the lifetime of an object with shared ownership by hand. If you find yourself doubting whether you can afford use of `std::shared_ptr`, reconsider whether you really need shared ownership. If exclusive ownership will do or even *may* do, `std::unique_ptr` is a better choice. Its performance profile is close to that for raw pointers, and "upgrading" from `std::unique_ptr` to `std::shared_ptr` is easy, because a `std::shared_ptr` can be created from a `std::unique_ptr`.

The reverse is not true. Once you've turned lifetime management of a resource over to a `std::shared_ptr`, there's no changing your mind. Even if the reference count is one, you can't reclaim ownership of the resource in order to, say, have a `std::unique_ptr` manage it. The ownership contract between a resource and the `std::shared_ptr`s that point to it is of the 'til-death-do-us-part variety. No divorce, no annulment, no dispensations.

Something else `std::shared_ptr`s can't do is work with arrays. In yet another difference from `std::unique_ptr`, `std::shared_ptr` has an API that's designed only for pointers to single objects. There's no `std::shared_ptr<T[]>`. From time to time, "clever" programmers stumble on the idea of using a `std::shared_ptr<T>` to point to an array, specifying a custom deleter to perform an array delete (i.e., `delete []`). This can be made to compile, but it's a horrible idea. For one thing, `std::shared_ptr` offers no `operator[]`, so indexing into the array requires awkward expressions based on pointer arithmetic. For another, `std::shared_ptr` supports derived-to-base pointer conversions that make sense for single objects, but that open holes in the type system when applied to arrays. (For this reason, the `std::unique_ptr<T[]>` API prohibits such conversions.) Most importantly, given the variety of C++11 alternatives to built-in arrays (e.g., `std::array`, `std::vector`, `std::string`), declaring a smart pointer to a dumb array is almost always a sign of bad design.

> **Things to Remember**
>
> - `std::shared_ptr`s offer convenience approaching that of garbage collection for the shared lifetime management of arbitrary resources.
> - Compared to `std::unique_ptr`, `std::shared_ptr` objects are typically twice as big, incur overhead for control blocks, and require atomic reference count manipulations.
> - Default resource destruction is via `delete`, but custom deleters are supported. The type of the deleter has no effect on the type of the `std::shared_ptr`.
> - Avoid creating `std::shared_ptr`s from variables of raw pointer type.

Item 20: Use `std::weak_ptr` for `std::shared_ptr`-like pointers that can dangle.

Paradoxically, it can be convenient to have a smart pointer that acts like a `std::shared_ptr` (see Item 19), but that doesn't participate in the shared ownership of the pointed-to resource. In other words, a pointer like `std::shared_ptr` that doesn't affect an object's reference count. This kind of smart pointer has to contend with a problem unknown to `std::shared_ptr`s: the possibility that what it points to has been destroyed. A truly smart pointer would deal with this problem by tracking when it *dangles*, i.e., when the object it is supposed to point to no longer exists. That's precisely the kind of smart pointer `std::weak_ptr` is.

You may be wondering how a `std::weak_ptr` could be useful. You'll probably wonder even more when you examine the `std::weak_ptr` API. It looks anything but smart. `std::weak_ptr`s can't be dereferenced, nor can they be tested for nullness. That's because `std::weak_ptr` isn't a standalone smart pointer. It's an augmentation of `std::shared_ptr`.

The relationship begins at birth. `std::weak_ptr`s are typically created from `std::shared_ptr`s. They point to the same place as the `std::shared_ptr`s initializing them, but they don't affect the reference count of the object they point to:

```
auto spw =                         // after spw is constructed,
  std::make_shared<Widget>();      // the pointed-to Widget's
```

```
                                  // ref count (RC) is 1. (See
                                  // Item 21 for info on
                                  // std::make_shared.)
…

std::weak_ptr<Widget> wpw(spw);   // wpw points to same Widget
                                  // as spw. RC remains 1
…

spw = nullptr;                    // RC goes to 0, and the
                                  // Widget is destroyed.
                                  // wpw now dangles
```

`std::weak_ptr`s that dangle are said to have *expired*. You can test for this directly,

```
if (wpw.expired()) …              // if wpw doesn't point
                                  // to an object…
```

but often what you desire is a check to see if a `std::weak_ptr` has expired and, if it hasn't (i.e., if it's not dangling), to access the object it points to. This is easier desired than done. Because `std::weak_ptr`s lack dereferencing operations, there's no way to write the code. Even if there were, separating the check and the dereference would introduce a race condition: between the call to `expired` and the dereferencing action, another thread might reassign or destroy the last `std::shared_ptr` pointing to the object, thus causing that object to be destroyed. In that case, your dereference would yield undefined behavior.

What you need is an atomic operation that checks to see if the `std::weak_ptr` has expired and, if not, gives you access to the object it points to. This is done by creating a `std::shared_ptr` from the `std::weak_ptr`. The operation comes in two forms, depending on what you'd like to have happen if the `std::weak_ptr` has expired when you try to create a `std::shared_ptr` from it. One form is `std::weak_ptr::lock`, which returns a `std::shared_ptr`. The `std::shared_ptr` is null if the `std::weak_ptr` has expired:

```
std::shared_ptr<Widget> spw1 = wpw.lock();  // if wpw's expired,
                                            // spw1 is null

auto spw2 = wpw.lock();                     // same as above,
                                            // but uses auto
```

The other form is the `std::shared_ptr` constructor taking a `std::weak_ptr` as an argument. In this case, if the `std::weak_ptr` has expired, an exception is thrown:

```
std::shared_ptr<Widget> spw3(wpw);    // if wpw's expired,
                                      // throw std::bad_weak_ptr
```

But you're probably still wondering about how `std::weak_ptr`s can be useful. Consider a factory function that produces smart pointers to read-only objects based on a unique ID. In accord with Item 18's advice regarding factory function return types, it returns a `std::unique_ptr`:

```
std::unique_ptr<const Widget> loadWidget(WidgetID id);
```

If `loadWidget` is an expensive call (e.g., because it performs file or database I/O) and it's common for IDs to be used repeatedly, a reasonable optimization would be to write a function that does what `loadWidget` does, but also caches its results. Clogging the cache with every `Widget` that has ever been requested can lead to performance problems of its own, however, so another reasonable optimization would be to destroy cached `Widget`s when they're no longer in use.

For this caching factory function, a `std::unique_ptr` return type is not a good fit. Callers should certainly receive smart pointers to cached objects, and callers should certainly determine the lifetime of those objects, but the cache needs a pointer to the objects, too. The cache's pointers need to be able to detect when they dangle, because when factory clients are finished using an object returned by the factory, that object will be destroyed, and the corresponding cache entry will dangle. The cached pointers should therefore be `std::weak_ptr`s—pointers that can detect when they dangle. That means that the factory's return type should be a `std::shared_ptr`, because `std::weak_ptr`s can detect when they dangle only when an object's lifetime is managed by `std::shared_ptr`s.

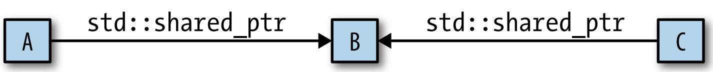
Here's a quick-and-dirty implementation of a caching version of `loadWidget`:

```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
{
  static std::unordered_map<WidgetID,
                            std::weak_ptr<const Widget>> cache;

  auto objPtr = cache[id].lock();   // objPtr is std::shared_ptr
                                    // to cached object (or null
                                    // if object's not in cache)

  if (!objPtr) {                    // if not in cache,
    objPtr = loadWidget(id);        // load it
    cache[id] = objPtr;             // cache it
  }
  return objPtr;
}
```

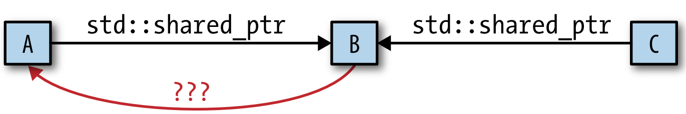
This implementation employs one of C++11's hash table containers (std::unordered_map), though it doesn't show the WidgetID hashing and equality-comparison functions that would also have to be present.

The implementation of fastLoadWidget ignores the fact that the cache may accumulate expired std::weak_ptrs corresponding to Widgets that are no longer in use (and have therefore been destroyed). The implementation can be refined, but rather than spend time on an issue that lends no additional insight into std::weak_ptrs, let's consider a second use case: the Observer design pattern. The primary components of this pattern are subjects (objects whose state may change) and observers (objects to be notified when state changes occur). In most implementations, each subject contains a data member holding pointers to its observers. That makes it easy for subjects to issue state change notifications. Subjects have no interest in controlling the lifetime of their observers (i.e., when they're destroyed), but they have a great interest in making sure that if an observer gets destroyed, subjects don't try to subsequently access it. A reasonable design is for each subject to hold a container of std::weak_ptrs to its observers, thus making it possible for the subject to determine whether a pointer dangles before using it.

As a final example of std::weak_ptr's utility, consider a data structure with objects A, B, and C in it, where A and C share ownership of B and therefore hold std::shared_ptrs to it:



Suppose it'd be useful to also have a pointer from B back to A. What kind of pointer should this be?



There are three choices:

- **A raw pointer.** With this approach, if A is destroyed, but C continues to point to B, B will contain a pointer to A that will dangle. B won't be able to detect that, so B may inadvertently dereference the dangling pointer. That would yield undefined behavior.
- **A std::shared_ptr.** In this design, A and B contain std::shared_ptrs to each other. The resulting std::shared_ptr cycle (A points to B and B points to A) will

prevent both A and B from being destroyed. Even if A and B are unreachable from other program data structures (e.g., because C no longer points to B), each will have a reference count of one. If that happens, A and B will have been leaked, for all practical purposes: it will be impossible for the program to access them, yet their resources will never be reclaimed.

- **A `std::weak_ptr`.** This avoids both problems above. If A is destroyed, B's pointer back to it will dangle, but B will be able to detect that. Furthermore, though A and B will point to one another, B's pointer won't affect A's reference count, hence can't keep A from being destroyed when `std::shared_ptr`s no longer point to it.

Using `std::weak_ptr` is clearly the best of these choices. However, it's worth noting that the need to employ `std::weak_ptr`s to break prospective cycles of `std::shared_ptr`s is not terribly common. In strictly hierarchal data structures such as trees, child nodes are typically owned only by their parents. When a parent node is destroyed, its child nodes should be destroyed, too. Links from parents to children are thus generally best represented by `std::unique_ptr`s. Back-links from children to parents can be safely implemented as raw pointers, because a child node should never have a lifetime longer than its parent. There's thus no risk of a child node dereferencing a dangling parent pointer.

Of course, not all pointer-based data structures are strictly hierarchical, and when that's the case, as well as in situations such as caching and the implementation of lists of observers, it's nice to know that `std::weak_ptr` stands at the ready.

From an efficiency perspective, the `std::weak_ptr` story is essentially the same as that for `std::shared_ptr`. `std::weak_ptr` objects are the same size as `std::shared_ptr` objects, they make use of the same control blocks as `std::shared_ptr`s (see Item 19), and operations such as construction, destruction, and assignment involve atomic reference count manipulations. That probably surprises you, because I wrote at the beginning of this Item that `std::weak_ptr`s don't participate in reference counting. Except that's not quite what I wrote. What I wrote was that `std::weak_ptr`s don't participate in the *shared ownership* of objects and hence don't affect the *pointed-to object's reference count*. There's actually a second reference count in the control block, and it's this second reference count that `std::weak_ptr`s manipulate. For details, continue on to Item 21.

Things to Remember

- Use `std::weak_ptr` for `std::shared_ptr`-like pointers that can dangle.
- Potential use cases for `std::weak_ptr` include caching, observer lists, and the prevention of `std::shared_ptr` cycles.

Item 21: Prefer `std::make_unique` and `std::make_shared` to direct use of `new`.

Let's begin by leveling the playing field for `std::make_unique` and `std::make_shared`. `std::make_shared` is part of C++11, but, sadly, `std::make_unique` isn't. It joined the Standard Library as of C++14. If you're using C++11, never fear, because a basic version of `std::make_unique` is easy to write yourself. Here, look:

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params)
{
  return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

As you can see, `make_unique` just perfect-forwards its parameters to the constructor of the object being created, constructs a `std::unique_ptr` from the raw pointer `new` produces, and returns the `std::unique_ptr` so created. This form of the function doesn't support arrays or custom deleters (see Item 18), but it demonstrates that with only a little effort, you can create `make_unique` if you need to.[3] Just remember not to put your version in namespace `std`, because you won't want it to clash with a vendor-provided version when you upgrade to a C++14 Standard Library implementation.

`std::make_unique` and `std::make_shared` are two of the three *`make` functions*: functions that take an arbitrary set of arguments, perfect-forward them to the constructor for a dynamically allocated object, and return a smart pointer to that object. The third `make` function is `std::allocate_shared`. It acts just like `std::make_shared`, except its first argument is an allocator object to be used for the dynamic memory allocation.

3 To create a full-featured `make_unique` with the smallest effort possible, search for the standardization document that gave rise to it, then copy the implementation you'll find there. The document you want is N3656 by Stephan T. Lavavej, dated 2013-04-18.

Even the most trivial comparison of smart pointer creation using and not using a make function reveals the first reason why using such functions is preferable. Consider:

```
auto upw1(std::make_unique<Widget>());      // with make func

std::unique_ptr<Widget> upw2(new Widget);   // without make func


auto spw1(std::make_shared<Widget>());      // with make func

std::shared_ptr<Widget> spw2(new Widget);   // without make func
```

I've highlighted the essential difference: the versions using new repeat the type being created, but the make functions don't. Repeating types runs afoul of a key tenet of software engineering: code duplication should be avoided. Duplication in source code increases compilation times, can lead to bloated object code, and generally renders a code base more difficult to work with. It often evolves into inconsistent code, and inconsistency in a code base often leads to bugs. Besides, typing something twice takes more effort than typing it once, and who's not a fan of reducing their typing burden?

The second reason to prefer make functions has to do with exception safety. Suppose we have a function to process a Widget in accord with some priority:

```
void processWidget(std::shared_ptr<Widget> spw, int priority);
```

Passing the std::shared_ptr by value may look suspicious, but Item 41 explains that if processWidget always makes a copy of the std::shared_ptr (e.g., by storing it in a data structure tracking Widgets that have been processed), this can be a reasonable design choice.

Now suppose we have a function to compute the relevant priority,

```
int computePriority();
```

and we use that in a call to processWidget that uses new instead of std::make_shared:

```
processWidget(std::shared_ptr<Widget>(new Widget),  // potential
              computePriority());                   // resource
                                                    // leak!
```

As the comment indicates, this code could leak the Widget conjured up by new. But how? Both the calling code and the called function are using std::shared_ptrs, and std::shared_ptrs are designed to prevent resource leaks. They automatically destroy what they point to when the last `std::shared_ptr` pointing there goes away. If everybody is using `std::shared_ptr`s everywhere, how can this code leak?

The answer has to do with compilers' translation of source code into object code. At runtime, the arguments for a function must be evaluated before the function can be invoked, so in the call to `processWidget`, the following things must occur before `processWidget` can begin execution:

- The expression "`new Widget`" must be evaluated, i.e., a `Widget` must be created on the heap.
- The constructor for the `std::shared_ptr<Widget>` responsible for managing the pointer produced by `new` must be executed.
- `computePriority` must run.

Compilers are not required to generate code that executes them in this order. "`new Widget`" must be executed before the `std::shared_ptr` constructor may be called, because the result of that `new` is used as an argument to that constructor, but `computePriority` may be executed before those calls, after them, or, crucially, *between* them. That is, compilers may emit code to execute the operations in this order:

1. Perform "`new Widget`".
2. Execute `computePriority`.
3. Run `std::shared_ptr` constructor.

If such code is generated and, at runtime, `computePriority` produces an exception, the dynamically allocated `Widget` from Step 1 will be leaked, because it will never be stored in the `std::shared_ptr` that's supposed to start managing it in Step 3.

Using `std::make_shared` avoids this problem. Calling code would look like this:

```
processWidget(std::make_shared<Widget>(),   // no potential
              computePriority());           // resource leak
```

At runtime, either `std::make_shared` or `computePriority` will be called first. If it's `std::make_shared`, the raw pointer to the dynamically allocated `Widget` is safely stored in the returned `std::shared_ptr` before `computePriority` is called. If `computePriority` then yields an exception, the `std::shared_ptr` destructor will see to it that the `Widget` it owns is destroyed. And if `computePriority` is called first and yields an exception, `std::make_shared` will not be invoked, and there will hence be no dynamically allocated `Widget` to worry about.

If we replace `std::shared_ptr` and `std::make_shared` with `std::unique_ptr` and `std::make_unique`, exactly the same reasoning applies. Using `std::make_unique` instead of `new` is thus just as important in writing exception-safe code as using `std::make_shared`.

A special feature of `std::make_shared` (compared to direct use of `new`) is improved efficiency. Using `std::make_shared` allows compilers to generate smaller, faster code that employs leaner data structures. Consider the following direct use of `new`:

```
std::shared_ptr<Widget> spw(new Widget);
```

It's obvious that this code entails a memory allocation, but it actually performs two. Item 19 explains that every `std::shared_ptr` points to a control block containing, among other things, the reference count for the pointed-to object. Memory for this control block is allocated in the `std::shared_ptr` constructor. Direct use of `new`, then, requires one memory allocation for the `Widget` and a second allocation for the control block.

If `std::make_shared` is used instead,

```
auto spw = std::make_shared<Widget>();
```

one allocation suffices. That's because `std::make_shared` allocates a single chunk of memory to hold both the `Widget` object and the control block. This optimization reduces the static size of the program, because the code contains only one memory allocation call, and it increases the speed of the executable code, because memory is allocated only once. Furthermore, using `std::make_shared` obviates the need for some of the bookkeeping information in the control block, potentially reducing the total memory footprint for the program.

The efficiency analysis for `std::make_shared` is equally applicable to `std::allocate_shared`, so the performance advantages of `std::make_shared` extend to that function, as well.

The arguments for preferring `make` functions over direct use of `new` are strong ones. Despite their software engineering, exception safety, and efficiency advantages, however, this Item's guidance is to *prefer* the `make` functions, not to rely on them exclusively. That's because there are circumstances where they can't or shouldn't be used.

For example, none of the `make` functions permit the specification of custom deleters (see Items 18 and 19), but both `std::unique_ptr` and `std::shared_ptr` have constructors that do. Given a custom deleter for a `Widget`,

```
auto widgetDeleter = [](Widget* pw) { … };
```

creating a smart pointer using it is straightforward using `new`:

```
std::unique_ptr<Widget, decltype(widgetDeleter)>
  upw(new Widget, widgetDeleter);

std::shared_ptr<Widget> spw(new Widget, widgetDeleter);
```

There's no way to do the same thing with a `make` function.

A second limitation of `make` functions stems from a syntactic detail of their implementations. Item 7 explains that when creating an object whose type overloads constructors both with and without `std::initializer_list` parameters, creating the object using braces prefers the `std::initializer_list` constructor, while creating the object using parentheses calls the non-`std::initializer_list` constructor. The `make` functions perfect-forward their parameters to an object's constructor, but do they do so using parentheses or using braces? For some types, the answer to this question makes a big difference. For example, in these calls,

```
auto upv = std::make_unique<std::vector<int>>(10, 20);

auto spv = std::make_shared<std::vector<int>>(10, 20);
```

do the resulting smart pointers point to `std::vector`s with 10 elements, each of value 20, or to `std::vector`s with two elements, one with value 10 and the other with value 20? Or is the result indeterminate?

The good news is that it's not indeterminate: both calls create `std::vector`s of size 10 with all values set to 20. That means that within the `make` functions, the perfect forwarding code uses parentheses, not braces. The bad news is that if you want to construct your pointed-to object using a braced initializer, you must use `new` directly. Using a `make` function would require the ability to perfect-forward a braced initializer, but, as Item 30 explains, braced initializers can't be perfect-forwarded. However, Item 30 also describes a workaround: use `auto` type deduction to create a `std::initializer_list` object from a braced initializer (see Item 2), then pass the `auto`-created object through the `make` function:

```
// create std::initializer_list
auto initList = { 10, 20 };

// create std::vector using std::initializer_list ctor
auto spv = std::make_shared<std::vector<int>>(initList);
```

For `std::unique_ptr`, these two scenarios (custom deleters and braced initializers) are the only ones where its `make` functions are problematic. For `std::shared_ptr` and its `make` functions, there are two more. Both are edge cases, but some developers live on the edge, and you may be one of them.

Some classes define their own versions of `operator new` and `operator delete`. The presence of these functions implies that the global memory allocation and deallocation routines for objects of these types are inappropriate. Often, class-specific routines are designed only to allocate and deallocate chunks of memory of precisely the size of objects of the class, e.g., `operator new` and `operator delete` for class `Widget` are often designed only to handle allocation and deallocation of chunks of memory of exactly size `sizeof(Widget)`. Such routines are a poor fit for `std::shared_ptr`'s support for custom allocation (via `std::allocate_shared`) and deallocation (via custom deleters), because the amount of memory that `std::allocate_shared` requests isn't the size of the dynamically allocated object, it's the size of that object *plus* the size of a control block. Consequently, using `make` functions to create objects of types with class-specific versions of `operator new` and `operator delete` is typically a poor idea.

The size and speed advantages of `std::make_shared` vis-à-vis direct use of `new` stem from `std::shared_ptr`'s control block being placed in the same chunk of memory as the managed object. When that object's reference count goes to zero, the object is destroyed (i.e., its destructor is called). However, the memory it occupies can't be released until the control block has also been destroyed, because the same chunk of dynamically allocated memory contains both.

As I noted, the control block contains bookkeeping information beyond just the reference count itself. The reference count tracks how many `std::shared_ptr`s refer to the control block, but the control block contains a second reference count, one that tallies how many `std::weak_ptr`s refer to the control block. This second reference count is known as the *weak count*.[4] When a `std::weak_ptr` checks to see if it has expired (see Item 19), it does so by examining the reference count (not the weak count) in the control block that it refers to. If the reference count is zero (i.e., if the pointed-to object has no `std::shared_ptr`s referring to it and has thus been destroyed), the `std::weak_ptr` has expired. Otherwise, it hasn't.

As long as `std::weak_ptr`s refer to a control block (i.e., the weak count is greater than zero), that control block must continue to exist. And as long as a control block exists, the memory containing it must remain allocated. The memory allocated by a `std::shared_ptr` `make` function, then, can't be deallocated until the last `std::shared_ptr` *and* the last `std::weak_ptr` referring to it have been destroyed.

4 In practice, the value of the weak count isn't always equal to the number of `std::weak_ptr`s referring to the control block, because library implementers have found ways to slip additional information into the weak count that facilitate better code generation. For purposes of this Item, we'll ignore this and assume that the weak count's value is the number of `std::weak_ptr`s referring to the control block.

If the object type is quite large and the time between destruction of the last `std::shared_ptr` and the last `std::weak_ptr` is significant, a lag can occur between when an object is destroyed and when the memory it occupied is freed:

```
class ReallyBigType { … };

auto pBigObj =                          // create very large
  std::make_shared<ReallyBigType>();    // object via
                                        // std::make_shared

…            // create std::shared_ptrs and std::weak_ptrs to
             // large object, use them to work with it

…            // final std::shared_ptr to object destroyed here,
             // but std::weak_ptrs to it remain

…            // during this period, memory formerly occupied
             // by large object remains allocated

…            // final std::weak_ptr to object destroyed here;
             // memory for control block and object is released
```

With a direct use of `new`, the memory for the `ReallyBigType` object can be released as soon as the last `std::shared_ptr` to it is destroyed:

```
class ReallyBigType { … };              // as before

std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);
                                        // create very large
                                        // object via new

…            // as before, create std::shared_ptrs and
             // std::weak_ptrs to object, use them with it

…            // final std::shared_ptr to object destroyed here,
             // but std::weak_ptrs to it remain;
             // memory for object is deallocated

…            // during this period, only memory for the
             // control block remains allocated

…            // final std::weak_ptr to object destroyed here;
             // memory for control block is released
```

Should you find yourself in a situation where use of `std::make_shared` is impossible or inappropriate, you'll want to guard yourself against the kind of exception-safety problems we saw earlier. The best way to do that is to make sure that when you use `new` directly, you immediately pass the result to a smart pointer constructor in *a statement that does nothing else*. This prevents compilers from generating code that could emit an exception between the use of `new` and invocation of the constructor for the smart pointer that will manage the `new`ed object.

As an example, consider a minor revision to the exception-unsafe call to the `processWidget` function we examined earlier. This time, we'll specify a custom deleter:

```
void processWidget(std::shared_ptr<Widget> spw,  // as before
                   int priority);

void cusDel(Widget *ptr);                        // custom
                                                 // deleter
```

Here's the exception-unsafe call:

```
processWidget(                                   // as before,
  std::shared_ptr<Widget>(new Widget, cusDel),   // potential
  computePriority()                              // resource
);                                               // leak!
```

Recall: if `computePriority` is called after "`new Widget`" but before the `std::shared_ptr` constructor, and if `computePriority` yields an exception, the dynamically allocated `Widget` will be leaked.

Here the use of a custom deleter precludes use of `std::make_shared`, so the way to avoid the problem is to put the allocation of the `Widget` and the construction of the `std::shared_ptr` into their own statement, then call `processWidget` with the resulting `std::shared_ptr`. Here's the essence of the technique, though, as we'll see in a moment, we can tweak it to improve its performance:

```
std::shared_ptr<Widget> spw(new Widget, cusDel);

processWidget(spw, computePriority());     // correct, but not
                                           // optimal; see below
```

This works, because a `std::shared_ptr` assumes ownership of the raw pointer passed to its constructor, even if that constructor yields an exception. In this example, if `spw`'s constructor throws an exception (e.g., due to an inability to dynamically allocate memory for a control block), it's still guaranteed that `cusDel` will be invoked on the pointer resulting from "`new Widget`".

The minor performance hitch is that in the exception-unsafe call, we're passing an rvalue to `processWidget`,

```
processWidget(
  std::shared_ptr<Widget>(new Widget, cusDel),  // arg is rvalue
  computePriority()
);
```

but in the exception-safe call, we're passing an lvalue:

```
processWidget(spw, computePriority());          // arg is lvalue
```

Because `processWidget`'s `std::shared_ptr` parameter is passed by value, construction from an rvalue entails only a move, while construction from an lvalue requires a copy. For `std::shared_ptr`, the difference can be significant, because copying a `std::shared_ptr` requires an atomic increment of its reference count, while moving a `std::shared_ptr` requires no reference count manipulation at all. For the exception-safe code to achieve the level of performance of the exception-unsafe code, we need to apply `std::move` to `spw` to turn it into an rvalue (see Item 23):

```
processWidget(std::move(spw),               // both efficient and
              computePriority());           // exception safe
```

That's interesting and worth knowing, but it's also typically irrelevant, because you'll rarely have a reason not to use a `make` function. And unless you have a compelling reason for doing otherwise, using a `make` function is what you should do.

Things to Remember

- Compared to direct use of `new`, `make` functions eliminate source code duplication, improve exception safety, and, for `std::make_shared` and `std::allocate_shared`, generate code that's smaller and faster.
- Situations where use of `make` functions is inappropriate include the need to specify custom deleters and a desire to pass braced initializers.
- For `std::shared_ptr`s, additional situations where `make` functions may be ill-advised include (1) classes with custom memory management and (2) systems with memory concerns, very large objects, and `std::weak_ptr`s that outlive the corresponding `std::shared_ptr`s.

Item 22: When using the Pimpl Idiom, define special member functions in the implementation file.

If you've ever had to combat excessive build times, you're familiar with the *Pimpl* ("pointer to implementation") *Idiom*. That's the technique whereby you replace the data members of a class with a pointer to an implementation class (or struct), put the data members that used to be in the primary class into the implementation class, and access those data members indirectly through the pointer. For example, suppose `Widget` looks like this:

```
class Widget {                       // in header "widget.h"
public:
  Widget();
  …
private:
  std::string name;
  std::vector<double> data;
  Gadget g1, g2, g3;                 // Gadget is some user-
};                                   // defined type
```

Because `Widget`'s data members are of types `std::string`, `std::vector`, and `Gadget`, headers for those types must be present for `Widget` to compile, and that means that `Widget` clients must `#include <string>`, `<vector>`, and `gadget.h`. Those headers increase the compilation time for `Widget` clients, plus they make those clients dependent on the contents of the headers. If a header's content changes, `Widget` clients must recompile. The standard headers `<string>` and `<vector>` don't change very often, but it could be that `gadget.h` is subject to frequent revision.

Applying the Pimpl Idiom in C++98 could have `Widget` replace its data members with a raw pointer to a struct that has been declared, but not defined:

```
class Widget {                   // still in header "widget.h"
public:
  Widget();
  ~Widget();                     // dtor is needed—see below
  …

private:
  struct Impl;                   // declare implementation struct
  Impl *pImpl;                   // and pointer to it
};
```

Because `Widget` no longer mentions the types `std::string`, `std::vector`, and `Gadget`, `Widget` clients no longer need to `#include` the headers for these types. That speeds compilation, and it also means that if something in these headers changes, `Widget` clients are unaffected.

A type that has been declared, but not defined, is known as an *incomplete type*. `Widget::Impl` is such a type. There are very few things you can do with an incomplete type, but declaring a pointer to it is one of them. The Pimpl Idiom takes advantage of that.

Part 1 of the Pimpl Idiom is the declaration of a data member that's a pointer to an incomplete type. Part 2 is the dynamic allocation and deallocation of the object that holds the data members that used to be in the original class. The allocation and deallocation code goes in the implementation file, e.g., for `Widget`, in `widget.cpp`:

```
#include "widget.h"            // in impl. file "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl {          // definition of Widget::Impl
  std::string name;            // with data members formerly
  std::vector<double> data;    // in Widget
  Gadget g1, g2, g3;
};

Widget::Widget()               // allocate data members for
: pImpl(new Impl)              // this Widget object
{}

Widget::~Widget()              // destroy data members for
{ delete pImpl; }              // this object
```

Here I'm showing `#include` directives to make clear that the overall dependencies on the headers for `std::string`, `std::vector`, and `Gadget` continue to exist. However, these dependencies have been moved from `widget.h` (which is visible to and used by `Widget` clients) to `widget.cpp` (which is visible to and used only by the `Widget` implementer). I've also highlighted the code that dynamically allocates and deallocates the `Impl` object. The need to deallocate this object when a `Widget` is destroyed is what necessitates the `Widget` destructor.

But I've shown you C++98 code, and that reeks of a bygone millennium. It uses raw pointers and raw `new` and raw `delete` and it's all just so…raw. This chapter is built on the idea that smart pointers are preferable to raw pointers, and if what we want is to dynamically allocate a `Widget::Impl` object inside the `Widget` constructor and have it destroyed at the same time the `Widget` is, `std::unique_ptr` (see Item 18) is precisely the tool we need. Replacing the raw `pImpl` pointer with a `std::unique_ptr` yields this code for the header file,

```
class Widget {                      // in "widget.h"
public:
  Widget();
  …

private:
```