3 books in 1

Effective C++ Digital Collection

140 Ways to Improve Your Programming

Scott Meyers

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Effective C++ Digital Collection

140 Ways to Improve Your Programming

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Note from the Publisher

The *Effective C++ Digital Collection* includes three bestselling C++ eBooks:

- *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*
- *More Effective C++: 35 New Ways to Improve Your Programs and Designs*
- *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*

By combining these seminal works into one eBook collection, you have easy access to the pragmatic, down-to-earth advice and proven wisdom that Scott Meyers is known for. This collection is essential reading for anyone working with C++ and STL.

To simplify access, we've appended "A" to pages of *Effective C++*, "B" to pages of *More Effective C++*, and "C" to pages of *Effective STL*. This enabled us to produce a single, comprehensive table of contents and dedicated indexes so that you can easily link to the topics you want and navigate between the books. We hope you find this collection useful!

—The editorial and production teams at Addison-Wesley

ISBN-13: 978-0-13-297919-1
ISBN-10: 0-13-297919-5

Second release, September 2012

This page intentionally left blank

Contents

Effective C++

More Effective C++

Effective STL

Effective C++
Third Edition

55 Specific Ways to Improve Your Programs and Designs

Scott Meyers

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Effective C++

Third Edition

Praise for *Effective C++, Third Edition*

"Scott Meyers' book, *Effective C++, Third Edition*, is distilled programming experience — experience that you would otherwise have to learn the hard way. This book is a great resource that I recommend to everybody who writes C++ professionally."

— *Peter Dulimov, ME, Engineer, Ranges and Assessing Unit, NAVSYSCOM, Australia*

"The third edition is still the best book on how to put all of the pieces of C++ together in an efficient, cohesive manner. If you claim to be a C++ programmer, you must read this book."

— *Eric Nagler, Consultant, Instructor, and author of* Learning C++

"The first edition of this book ranks among the small (very small) number of books that I credit with significantly elevating my skills as a 'professional' software developer. Like the others, it was practical and easy to read, but loaded with important advice. *Effective C++, Third Edition*, continues that tradition. C++ is a very powerful programming language. If C gives you enough rope to hang yourself, C++ is a hardware store with lots of helpful people ready to tie knots for you. Mastering the points discussed in this book will definitely increase your ability to effectively use C++ and reduce your stress level."

— *Jack W. Reeves, Chief Executive Officer, Bleading Edge Software Technologies*

"Every new developer joining my team has one assignment — to read this book."

— *Michael Lanzetta, Senior Software Engineer*

"I read the first edition of *Effective C++* about nine years ago, and it immediately became my favorite book on C++. In my opinion, *Effective C++, Third Edition*, remains a mustread today for anyone who wishes to program effectively in C++. We would live in a better world if C++ programmers had to read this book before writing their first line of professional C++ code."

— *Danny Rabbani, Software Development Engineer*

"I encountered the first edition of Scott Meyers' *Effective C++* as a struggling programmer in the trenches, trying to get better at what I was doing. What a lifesaver! I found Meyers' advice was practical, useful, and effective, fulfilling the promise of the title 100 percent. The third edition brings the practical realities of using C++ in serious development projects right up to date, adding chapters on the language's very latest issues and features. I was delighted to still find myself learning something interesting and new from the latest edition of a book I already thought I knew well."

— *Michael Topic, Technical Program Manager*

"From Scott Meyers, the guru of C++, this is the definitive guide for anyone who wants to use C++ safely and effectively, or is transitioning from any other OO language to C++. This book has valuable information presented in a clear, concise, entertaining, and insightful manner."

— *Siddhartha Karan Singh, Software Developer*

"This should be the second book on C++ that any developer should read, after a general introductory text. It goes beyond the *how* and *what* of C++ to address the *why* and *wherefore*. It helped me go from knowing the syntax to understanding the philosophy of C++ programming."

— *Timothy Knox, Software Developer*

"This is a fantastic update of a classic C++ text. Meyers covers a lot of new ground in this volume, and every serious C++ programmer should have a copy of this new edition."

— *Jeffrey Somers, Game Programmer*

"*Effective C++, Third Edition*, covers the things you should be doing when writing code and does a terrific job of explaining why those things are important. Think of it as best practices for writing C++."

— *Jeff Scherpelz, Software Development Engineer*

"As C++ embraces change, Scott Meyers' *Effective C++, Third Edition*, soars to remain in perfect lock-step with the language. There are many fine introductory books on C++, but exactly one *second* book stands head and shoulders above the rest, and you're holding it. With Scott guiding the way, prepare to do some soaring of your own!"

— *Leor Zolman, C++ Trainer and Pundit, BD Software*

"This book is a must-have for both C++ veterans and newbies. After you have finished reading it, it will not collect dust on your bookshelf — you will refer to it all the time."

— *Sam Lee, Software Developer*

"Reading this book transforms ordinary C++ programmers into expert C++ programmers, step-by-step, using 55 easy-to-read items, each describing one technique or tip."

— *Jeffrey D. Oldham, Ph.D., Software Engineer, Google*

"Scott Meyers' *Effective C++* books have long been required reading for new and experienced C++ programmers alike. This new edition, incorporating almost a decade's worth of C++ language development, is his most content-packed book yet. He does not merely describe the problems inherent in the language, but instead he provides unambiguous and easy-to-follow advice on how to avoid the pitfalls and write 'effective C++.' I expect every C++ programmer to have read it."

— *Philipp K. Janert, Ph.D., Software Development Manager*

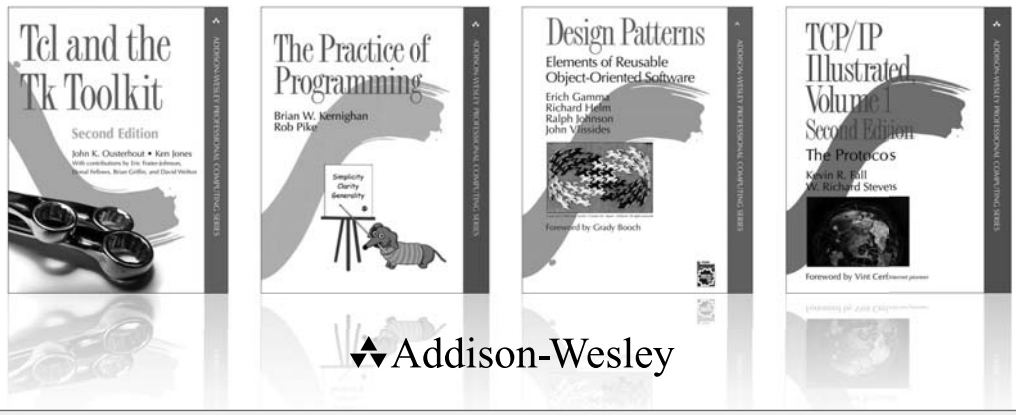
"Each previous edition of *Effective C++* has been the must-have book for developers who have used C++ for a few months or a few years, long enough to stumble into the traps latent in this rich language. In this third edition, Scott Meyers extensively refreshes his sound advice for the modern world of new language and library features and the programming styles that have evolved to use them. Scott's engaging writing style makes it easy to assimilate his guidelines on your way to becoming an effective C++ developer."

— *David Smallberg, Instructor, DevelopMentor; Lecturer, Computer Science, UCLA*

"*Effective C++* has been completely updated for twenty-first-century C++ practice and can continue to claim to be the first *second* book for all C++ practitioners."

— *Matthew Wilson, Ph.D., author of* Imperfect C++

The Addison-Wesley Professional Computing Series



Visit informit.com/series/professionalcomputing
for a complete list of available publications.

The Addison-Wesley Professional Computing Series was created in 1990 to provide serious programmers and networking professionals with well-written and practical reference books. There are few places to turn for accurate and authoritative books on current and cutting-edge technology. We hope that our books will help you understand the state of the art in programming languages, operating systems, and networks.

Consulting Editor Brian W. Kernighan

Make sure to connect with us!
informit.com/socialconnect

Addison Wesley | informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE | Safari Books Online

Effective C++

Third Edition

55 Specific Ways to Improve Your Programs and Designs

Scott Meyers

ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Control Number: 2005924388

ISBN 0-321-33487-6

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

First printing, May 2005

For Nancy,
without whom nothing
would be much worth doing

Wisdom and beauty form a very rare combination.

— Petronius Arbiter
Satyricon, XCIV

This page intentionally left blank

And in memory of Persephone,
1995–2004

This page intentionally left blank

Preface

I wrote the original edition of *Effective C++* in 1991. When the time came for a second edition in 1997, I updated the material in important ways, but, because I didn't want to confuse readers familiar with the first edition, I did my best to retain the existing structure: 48 of the original 50 Item titles remained essentially unchanged. If the book were a house, the second edition was the equivalent of freshening things up by replacing carpets, paint, and light fixtures.

For the third edition, I tore the place down to the studs. (There were times I wished I'd gone all the way to the foundation.) The world of C++ has undergone enormous change since 1991, and the goal of this book — to identify the most important C++ programming guidelines in a small, readable package — was no longer served by the Items I'd established nearly 15 years earlier. In 1991, it was reasonable to assume that C++ programmers came from a C background. Now, programmers moving to C++ are just as likely to come from Java or C#. In 1991, inheritance and object-oriented programming were new to most programmers. Now they're well-established concepts, and exceptions, templates, and generic programming are the areas where people need more guidance. In 1991, nobody had heard of design patterns. Now it's hard to discuss software systems without referring to them. In 1991, work had just begun on a formal standard for C++. Now that standard is eight years old, and work has begun on the next version.

To address these changes, I wiped the slate as clean as I could and asked myself, "What are the most important pieces of advice for practicing C++ programmers in 2005?" The result is the set of Items in this new edition. The book has new chapters on resource management and on programming with templates. In fact, template concerns are woven throughout the text, because they affect almost everything in C++. The book also includes new material on programming in the presence of exceptions, on applying design patterns, and on using the

new TR1 library facilities. (TR1 is described in Item 54.) It acknowledges that techniques and approaches that work well in single-threaded systems may not be appropriate in multithreaded systems. Well over half the material in the book is new. However, most of the fundamental information in the second edition continues to be important, so I found a way to retain it in one form or another. (You'll find a mapping between the second and third edition Items in Appendix B.)

I've worked hard to make this book as good as I can, but I have no illusions that it's perfect. If you feel that some of the Items in this book are inappropriate as general advice; that there is a better way to accomplish a task examined in the book; or that one or more of the technical discussions is unclear, incomplete, or misleading, please tell me. If you find an error of any kind — technical, grammatical, typographical, *whatever* — please tell me that, too. I'll gladly add to the acknowledgments in later printings the name of the first person to bring each problem to my attention.

Even with the number of Items expanded to 55, the set of guidelines in this book is far from exhaustive. But coming up with good rules — ones that apply to almost all applications almost all the time — is harder than it might seem. If you have suggestions for additional guidelines, I would be delighted to hear about them.

I maintain a list of changes to this book since its first printing, including bug fixes, clarifications, and technical updates. The list is available at the *Effective C++ Errata* web page, http://aristeia.com/BookErrata/ec++3e-errata.html. If you'd like to be notified when I update the list, I encourage you to join my mailing list. I use it to make announcements likely to interest people who follow my professional work. For details, consult http://aristeia.com/MailingList/.

SCOTT DOUGLAS MEYERS
http://aristeia.com/

STAFFORD, OREGON
APRIL 2005

Acknowledgments

Effective C++ has existed for fifteen years, and I started learning C++ about three years before I wrote the book. The "*Effective C++* project" has thus been under development for nearly two decades. During that time, I have benefited from the insights, suggestions, corrections, and, occasionally, dumbfounded stares of hundreds (thousands?) of people. Each has helped improve *Effective C++*. I am grateful to them all.

I've given up trying to keep track of where I learned what, but one general source of information has helped me as long as I can remember: the Usenet C++ newsgroups, especially comp.lang.c++.moderated and comp.std.c++. Many of the Items in this book — perhaps most — have benefited from the vetting of technical ideas at which the participants in these newsgroups excel.

Regarding new material in the third edition, Steve Dewhurst worked with me to come up with an initial set of candidate Items. In Item 11, the idea of implementing operator= via copy-and-swap came from Herb Sutter's writings on the topic, e.g., Item 13 of his *Exceptional C++* (Addison-Wesley, 2000). RAII (see Item 13) is from Bjarne Stroustrup's *The C++ Programming Language* (Addison-Wesley, 2000). The idea behind Item 17 came from the "Best Practices" section of the Boost shared_ptr web page, http://boost.org/libs/smart_ptr/shared_ptr.htm#BestPractices and was refined by Item 21 of Herb Sutter's *More Exceptional C++* (Addison-Wesley, 2002). Item 29 was strongly influenced by Herb Sutter's extensive writings on the topic, e.g., Items 8-19 of *Exceptional C++*, Items 17–23 of *More Exceptional C++*, and Items 11–13 of *Exceptional C++ Style* (Addison-Wesley, 2005); David Abrahams helped me better understand the three exception safety guarantees. The NVI idiom in Item 35 is from Herb Sutter's column, "Virtuality," in the September 2001 *C/C++ Users Journal*. In that same Item, the Template Method and Strategy design patterns are from *Design Patterns* (Addison-Wesley, 1995) by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The idea of using the NVI idiom in Item 37 came

from Hendrik Schober. David Smallberg contributed the motivation for writing a custom set implementation in Item 38. Item 39's observation that the EBO generally isn't available under multiple inheritance is from David Vandevoorde's and Nicolai M. Josuttis' *C++ Templates* (Addison-Wesley, 2003). In Item 42, my initial understanding about typename came from Greg Comeau's C++ and C FAQ (http://www.comeaucomputing.com/techtalk/#typename), and Leor Zolman helped me realize that my understanding was incorrect. (My fault, not Greg's.) The essence of Item 46 is from Dan Saks' talk, "Making New Friends." The idea at the end of Item 52 that if you declare one version of operator new, you should declare them all, is from Item 22 of Herb Sutter's *Exceptional C++ Style*. My understanding of the Boost review process (summarized in Item 55) was refined by David Abrahams.

Everything above corresponds to who or where *I* learned about something, not necessarily to who or where the thing was invented or first published.

My notes tell me that I also used information from Steve Clamage, Antoine Trux, Timothy Knox, and Mike Kaelbling, though, regrettably, the notes fail to tell me how or where.

Drafts of the first edition were reviewed by Tom Cargill, Glenn Carroll, Tony Davis, Brian Kernighan, Jak Kirman, Doug Lea, Moises Lejter, Eugene Santos, Jr., John Shewchuk, John Stasko, Bjarne Stroustrup, Barbara Tilly, and Nancy L. Urbano. I received suggestions for improvements that I was able to incorporate in later printings from Nancy L. Urbano, Chris Treichel, David Corbin, Paul Gibson, Steve Vinoski, Tom Cargill, Neil Rhodes, David Bern, Russ Williams, Robert Brazile, Doug Morgan, Uwe Steinmüller, Mark Somer, Doug Moore, David Smallberg, Seth Meltzer, Oleg Shteynbuk, David Papurt, Tony Hansen, Peter McCluskey, Stefan Kuhlins, David Braunegg, Paul Chisholm, Adam Zell, Clovis Tondo, Mike Kaelbling, Natraj Kini, Lars Nyman, Greg Lutz, Tim Johnson, John Lakos, Roger Scott, Scott Frohman, Alan Rooks, Robert Poor, Eric Nagler, Antoine Trux, Cade Roux, Chandrika Gokul, Randy Mangoba, and Glenn Teitelbaum.

Drafts of the second edition were reviewed by Derek Bosch, Tim Johnson, Brian Kernighan, Junichi Kimura, Scott Lewandowski, Laura Michaels, David Smallberg, Clovis Tondo, Chris Van Wyk, and Oleg Zabluda. Later printings benefited from comments from Daniel Steinberg, Arunprasad Marathe, Doug Stapp, Robert Hall, Cheryl Ferguson, Gary Bartlett, Michael Tamm, Kendall Beaman, Eric Nagler, Max Hailperin, Joe Gottman, Richard Weeks, Valentin Bonnard, Jun He, Tim King, Don Maier, Ted Hill, Mark Harrison, Michael Rubenstein, Mark Rodgers, David Goh, Brenton Cooper, Andy Thomas-Cramer,

Antoine Trux, John Wait, Brian Sharon, Liam Fitzpatrick, Bernd Mohr, Gary Yee, John O'Hanley, Brady Patterson, Christopher Peterson, Feliks Kluzniak, Isi Dunietz, Christopher Creutzi, Ian Cooper, Carl Harris, Mark Stickel, Clay Budin, Panayotis Matsinopoulos, David Smallberg, Herb Sutter, Pajo Misljencevic, Giulio Agostini, Fredrik Blomqvist, Jimmy Snyder, Byrial Jensen, Witold Kuzminski, Kazunobu Kuriyama, Michael Christensen, Jorge Yáñez Teruel, Mark Davis, Marty Rabinowitz, Ares Lagae, and Alexander Medvedev.

An early partial draft of this edition was reviewed by Brian Kernighan, Angelika Langer, Jesse Laeuchli, Roger E. Pedersen, Chris Van Wyk, Nicholas Stroustrup, and Hendrik Schober. Reviewers for a full draft were Leor Zolman, Mike Tsao, Eric Nagler, Gene Gutnik, David Abrahams, Gerhard Kreuzer, Drosos Kourounis, Brian Kernighan, Andrew Kirmse, Balog Pal, Emily Jagdhar, Eugene Kalenkovich, Mike Roze, Enrico Carrara, Benjamin Berck, Jack Reeves, Steve Schirripa, Martin Fallenstedt, Timothy Knox, Yun Bai, Michael Lanzetta, Philipp Janert, Guido Bartolucci, Michael Topic, Jeff Scherpelz, Chris Nauroth, Nishant Mittal, Jeff Somers, Hal Moroff, Vincent Manis, Brandon Chang, Greg Li, Jim Meehan, Alan Geller, Siddhartha Singh, Sam Lee, Sasan Dashtinezhad, Alex Marin, Steve Cai, Thomas Fruchterman, Cory Hicks, David Smallberg, Gunavardhan Kakulapati, Danny Rabbani, Jake Cohen, Hendrik Schober, Paco Viciana, Glenn Kennedy, Jeffrey D. Oldham, Nicholas Stroustrup, Matthew Wilson, Andrei Alexandrescu, Tim Johnson, Leon Matthews, Peter Dulimov, and Kevlin Henney. Drafts of some individual Items were reviewed by Herb Sutter and Attila F. Feher.

Reviewing an unpolished (possibly incomplete) manuscript is demanding work, and doing it under time pressure only makes it harder. I continue to be grateful that so many people have been willing to undertake it for me.

Reviewing is harder still if you have no background in the material being discussed and are expected to catch *every* problem in the manuscript. Astonishingly, some people still choose to be copy editors. Chrysta Meadowbrooke was the copy editor for this book, and her very thorough work exposed many problems that eluded everyone else.

Leor Zolman checked all the code examples against multiple compilers in preparation for the full review, then did it again after I revised the manuscript. If any errors remain, I'm responsible for them, not Leor.

Karl Wiegers and especially Tim Johnson offered rapid, helpful feedback on back cover copy.

Since publication of the first printing, I have incorporated revisions suggested by Jason Ross, Robert Yokota, Bernhard Merkle, Attila Feher, Gerhard Kreuzer, Marcin Sochacki, J. Daniel Smith, Idan Lupinsky, G. Wade Johnson, Clovis Tondo, Joshua Lehrer, T. David Hudson, Phillip Hellewell, Thomas Schell, Eldar Ronen, Ken Kobayashi, Cameron Mac Minn, John Hershberger, Alex Dumov, Vincent Stojanov, Andrew Henrick, Jiongxiong Chen, Balbir Singh, Fraser Ross, Niels Dekker, Harsh Gaurav Vangani, Vasily Poshehonov, Yukitoshi Fujimura, Alex Howlett, Ed Ji Xihuang. Mike Rizzi, Balog Pal, David Solomon, Tony Oliver, Martin Rottinger, Miaohua, and Brian Johnson.

John Wait, my editor for the first two editions of this book, foolishly signed up for another tour of duty in that capacity. His assistant, Denise Mickelsen, adroitly handled my frequent pestering with a pleasant smile. (At least I think she's been smiling. I've never actually seen her.) Julie Nahil drew the short straw and hence became my production manager. She handled the overnight loss of six weeks in the production schedule with remarkable equanimity. John Fuller (her boss) and Marty Rabinowitz (his boss) helped out with production issues, too. Vanessa Moore's official job was to help with FrameMaker issues and PDF preparation, but she also added the entries to Appendix B and formatted it for printing on the inside cover. Solveig Haugland helped with index formatting. Sandra Schroeder and Chuti Prasertsith were responsible for cover design, though Chuti seems to have been the one who had to rework the cover each time I said, "But what about *this* photo with a stripe of *that* color...?" Chanda Leary-Coutu got tapped for the heavy lifting in marketing.

During the months I worked on the manuscript, the TV series *Buffy the Vampire Slayer* often helped me "de-stress" at the end of the day. Only with great restraint have I kept Buffyspeak out of the book.

Kathy Reed taught me programming in 1971, and I'm gratified that we remain friends to this day. Donald French hired me and Moises Lejter to create C++ training materials in 1989 (an act that led to my *really* knowing C++), and in 1991 he engaged me to present them at Stratus Computer. The students in that class encouraged me to write what ultimately became the first edition of this book. Don also introduced me to John Wait, who agreed to publish it.

My wife, Nancy L. Urbano, continues to encourage my writing, even after seven book projects, a CD adaptation, and a dissertation. She has unbelievable forbearance. I couldn't do what I do without her.

From start to finish, our dog, Persephone, has been a companion without equal. Sadly, for much of this project, her companionship has taken the form of an urn in the office. We really miss her.

Introduction

Learning the fundamentals of a programming language is one thing; learning how to design and implement *effective* programs in that language is something else entirely. This is especially true of C++, a language boasting an uncommon range of power and expressiveness. Properly used, C++ can be a joy to work with. An enormous variety of designs can be directly expressed and efficiently implemented. A judiciously chosen and carefully crafted set of classes, functions, and templates can make application programming easy, intuitive, efficient, and nearly error-free. It isn't unduly difficult to write effective C++ programs, *if* you know how to do it. Used without discipline, however, C++ can lead to code that is incomprehensible, unmaintainable, inextensible, inefficient, and just plain wrong.

The purpose of this book is to show you how to use C++ *effectively*. I assume you already know C++ as a *language* and that you have some experience in its use. What I provide here is a guide to using the language so that your software is comprehensible, maintainable, portable, extensible, efficient, and likely to behave as you expect.

The advice I proffer falls into two broad categories: general design strategies, and the nuts and bolts of specific language features. The design discussions concentrate on how to choose between different approaches to accomplishing something in C++. How do you choose between inheritance and templates? Between public and private inheritance? Between private inheritance and composition? Between member and non-member functions? Between pass-by-value and pass-by-reference? It's important to make these decisions correctly at the outset, because a poor choice may not become apparent until much later in the development process, at which point rectifying it is often difficult, time-consuming, and expensive.

Even when you know exactly what you want to do, getting things just right can be tricky. What's the proper return type for assignment operators? When should a destructor be virtual? How should operator

new behave when it can't find enough memory? It's crucial to sweat details like these, because failure to do so almost always leads to unexpected, possibly mystifying program behavior. This book will help you avoid that.

This is not a comprehensive reference for C++. Rather, it's a collection of 55 specific suggestions (I call them *Items*) for how you can improve your programs and designs. Each Item stands more or less on its own, but most also contain references to other Items. One way to read the book, then, is to start with an Item of interest, then follow its references to see where they lead you.

The book isn't an introduction to C++, either. In Chapter 2, for example, I'm eager to tell you all about the proper implementations of constructors, destructors, and assignment operators, but I assume you already know or can go elsewhere to find out what these functions do and how they are declared. A number of C++ books contain information such as that.

The purpose of *this* book is to highlight those aspects of C++ programming that are often overlooked. Other books describe the different parts of the language. This book tells you how to combine those parts so you end up with effective programs. Other books tell you how to get your programs to compile. This book tells you how to avoid problems that compilers won't tell you about.

At the same time, this book limits itself to *standard* C++. Only features in the official language standard have been used here. Portability is a key concern in this book, so if you're looking for platform-dependent hacks and kludges, this is not the place to find them.

Another thing you won't find in this book is the C++ Gospel, the One True Path to perfect C++ software. Each of the Items in this book provides guidance on how to develop better designs, how to avoid common problems, or how to achieve greater efficiency, but none of the Items is universally applicable. Software design and implementation is a complex task, one colored by the constraints of the hardware, the operating system, and the application, so the best I can do is provide *guidelines* for creating better programs.

If you follow all the guidelines all the time, you are unlikely to fall into the most common traps surrounding C++, but guidelines, by their nature, have exceptions. That's why each Item has an explanation. The explanations are the most important part of the book. Only by understanding the rationale behind an Item can you determine whether it applies to the software you are developing and to the unique constraints under which you toil.

The best use of this book is to gain insight into how C++ behaves, why it behaves that way, and how to use its behavior to your advantage. Blind application of the Items in this book is clearly inappropriate, but at the same time, you probably shouldn't violate any of the guidelines without a good reason.

Terminology

There is a small C++ vocabulary that every programmer should understand. The following terms are important enough that it is worth making sure we agree on what they mean.

A ***declaration*** tells compilers about the name and type of something, but it omits certain details. These are declarations:

```
extern int x;                                   // object declaration

std::size_t numDigits(int number);              // function declaration

class Widget;                                   // class declaration

template<typename T>                            // template declaration
class GraphNode;                                // (see Item 42 for info on
                                                // the use of "typename")
```

Note that I refer to the integer x as an "object," even though it's of built-in type. Some people reserve the name "object" for variables of user-defined type, but I'm not one of them. Also note that the function numDigits' return type is std::size_t, i.e., the type size_t in namespace std. That namespace is where virtually everything in C++'s standard library is located. However, because C's standard library (the one from C89, to be precise) can also be used in C++, symbols inherited from C (such as size_t) may exist at global scope, inside std, or both, depending on which headers have been #included. In this book, I assume that C++ headers have been #included, and that's why I refer to std::size_t instead of just size_t. When referring to components of the standard library in prose, I typically omit references to std, relying on you to recognize that things like size_t, vector, and cout are in std. In example code, I always include std, because real code won't compile without it.

size_t, by the way, is just a typedef for some unsigned type that C++ uses when counting things (e.g., the number of characters in a char*-based string, the number of elements in an STL container, etc.). It's also the type taken by the operator[] functions in vector, deque, and string, a convention we'll follow when defining our own operator[] functions in Item 3.

Each function's declaration reveals its ***signature***, i.e., its parameter and return types. A function's signature is the same as its type. In the

case of numDigits, the signature is std::size_t (int), i.e., "function taking an int and returning a std::size_t." The official C++ definition of "signature" excludes the function's return type, but in this book, it's more useful to have the return type be considered part of the signature.

A ***definition*** provides compilers with the details a declaration omits. For an object, the definition is where compilers set aside memory for the object. For a function or a function template, the definition provides the code body. For a class or a class template, the definition lists the members of the class or template:

```
int x;                                          // object definition

std::size_t numDigits(int number)               // function definition.
{                                               // (This function returns
  std::size_t digitsSoFar = 1;                  // the number of digits
                                                // in its parameter.)
  while ((number /= 10) != 0) ++digitsSoFar;

  return digitsSoFar;
}

class Widget {                                  // class definition
public:
  Widget();
  ~Widget();
  ...
};

template<typename T>                            // template definition
class GraphNode {
public:
  GraphNode();
  ~GraphNode();
  ...
};
```

Initialization is the process of giving an object its first value. For objects generated from structs and classes, initialization is performed by constructors. A ***default constructor*** is one that can be called without any arguments. Such a constructor either has no parameters or has a default value for every parameter:

```
class A {
public:
  A();                                  // default constructor
};

class B {
public:
  explicit B(int x = 0, bool b = true); // default constructor; see below
};                                      // for info on "explicit"
```

```
class C {
public:
  explicit C(int x);                    // not a default constructor
};
```

The constructors for classes B and C are declared explicit here. That prevents them from being used to perform implicit type conversions, though they may still be used for explicit type conversions:

```
void doSomething(B bObject);            // a function taking an object of
                                        // type B

B bObj1;                                // an object of type B

doSomething(bObj1);                     // fine, passes a B to doSomething

B bObj2(28);                            // fine, creates a B from the int 28
                                        // (the bool defaults to true)

doSomething(28);                        // error! doSomething takes a B,
                                        // not an int, and there is no
                                        // implicit conversion from int to B

doSomething(B(28));                     // fine, uses the B constructor to
                                        // explicitly convert (i.e., cast) the
                                        // int to a B for this call. (See
                                        // Item 27 for info on casting.)
```

Constructors declared explicit are usually preferable to non-explicit ones, because they prevent compilers from performing unexpected (often unintended) type conversions. Unless I have a good reason for allowing a constructor to be used for implicit type conversions, I declare it explicit. I encourage you to follow the same policy.

Please note how I've highlighted the cast in the example above. Throughout this book, I use such highlighting to call your attention to material that is particularly noteworthy. (I also highlight chapter numbers, but that's just because I think it looks nice.)

The ***copy constructor*** is used to initialize an object with a different object of the same type, and the ***copy assignment operator*** is used to copy the value from one object to another of the same type:

```
class Widget {
public:
  Widget();                             // default constructor
  Widget(const Widget& rhs);            // copy constructor
  Widget& operator=(const Widget& rhs); // copy assignment operator
  ...
};

Widget w1;                              // invoke default constructor

Widget w2(w1);                          // invoke copy constructor

w1 = w2;                                // invoke copy
                                        // assignment operator
```

Read carefully when you see what appears to be an assignment, because the "=" syntax can also be used to call the copy constructor:

```
Widget w3 = w2;                    // invoke copy constructor!
```

Fortunately, copy construction is easy to distinguish from copy assignment. If a new object is being defined (such as w3 in the statement above), a constructor has to be called; it can't be an assignment. If no new object is being defined (such as in the "w1 = w2" statement above), no constructor can be involved, so it's an assignment.

The copy constructor is a particularly important function, because it defines how an object is passed by value. For example, consider this:

```
bool hasAcceptableQuality(Widget w);
...
Widget aWidget;
if (hasAcceptableQuality(aWidget)) ...
```

The parameter w is passed to hasAcceptableQuality by value, so in the call above, aWidget is copied into w. The copying is done by Widget's copy constructor. Pass-by-value *means* "call the copy constructor." (However, it's generally a bad idea to pass user-defined types by value. Pass-by-reference-to-const is typically a better choice. For details, see Item 20.)

The ***STL*** is the Standard Template Library, the part of C++'s standard library devoted to containers (e.g., vector, list, set, map, etc.), iterators (e.g., vector<int>::iterator, set<string>::iterator, etc.), algorithms (e.g., for_each, find, sort, etc.), and related functionality. Much of that related functionality has to do with ***function objects***: objects that act like functions. Such objects come from classes that overload operator(), the function call operator. If you're unfamiliar with the STL, you'll want to have a decent reference available as you read this book, because the STL is too useful for me not to take advantage of it. Once you've used it a little, you'll feel the same way.

Programmers coming to C++ from languages like Java or C# may be surprised at the notion of ***undefined behavior***. For a variety of reasons, the behavior of some constructs in C++ is literally not defined: you can't reliably predict what will happen at runtime. Here are two examples of code with undefined behavior:

```
int *p = 0;                        // p is a null pointer
std::cout << *p;                   // dereferencing a null pointer
                                   // yields undefined behavior
```

```
char name[] = "Darla";          // name is an array of size 6 (don't
                                // forget the trailing null!)
char c = name[10];              // referring to an invalid array index
                                // yields undefined behavior
```

To emphasize that the results of undefined behavior are not predictable and may be very unpleasant, experienced C++ programmers often say that programs with undefined behavior can erase your hard drive. It's true: a program with undefined behavior *could* erase your hard drive. But it's not probable. More likely is that the program will behave erratically, sometimes running normally, other times crashing, still other times producing incorrect results. Effective C++ programmers do their best to steer clear of undefined behavior. In this book, I point out a number of places where you need to be on the lookout for it.

Another term that may confuse programmers coming to C++ from another language is ***interface***. Java and the .NET languages offer Interfaces as a language element, but there is no such thing in C++, though Item 31 discusses how to approximate them. When I use the term "interface," I'm generally talking about a function's signature, about the accessible elements of a class (e.g., a class's "public interface," "protected interface," or "private interface"), or about the expressions that must be valid for a template's type parameter (see Item 41). That is, I'm talking about interfaces as a fairly general design idea.

A ***client*** is someone or something that uses the code (typically the interfaces) you write. A function's clients, for example, are its users: the parts of the code that call the function (or take its address) as well as the humans who write and maintain such code. The clients of a class or a template are the parts of the software that use the class or template, as well as the programmers who write and maintain that code. When discussing clients, I typically focus on programmers, because programmers can be confused, misled, or annoyed by bad interfaces. The code they write can't be.

You may not be used to thinking about clients, but I'll spend a good deal of time trying to convince you to make their lives as easy as you can. After all, you are a client of the software other people develop. Wouldn't you want those people to make things easy for you? Besides, at some point you'll almost certainly find yourself in the position of being your own client (i.e., using code you wrote), and at that point, you'll be glad you kept client concerns in mind when developing your interfaces.

In this book, I often gloss over the distinction between functions and function templates and between classes and class templates. That's because what's true about one is often true about the other. In situations where this is not the case, I distinguish among classes, functions, and the templates that give rise to classes and functions.

When referring to constructors and destructors in code comments, I sometimes use the abbreviations ***ctor*** and ***dtor***.

Naming Conventions

I have tried to select meaningful names for objects, classes, functions, templates, etc., but the meanings behind some of my names may not be immediately apparent. Two of my favorite parameter names, for example, are lhs and rhs. They stand for "left-hand side" and "right-hand side," respectively. I often use them as parameter names for functions implementing binary operators, e.g., operator== and operator*. For example, if a and b are objects representing rational numbers, and if Rational objects can be multiplied via a non-member operator* function (as Item 24 explains is likely to be the case), the expression

```
a * b
```

is equivalent to the function call

```
operator*(a, b)
```

In Item 24, I declare operator* like this:

```
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

As you can see, the left-hand operand, a, is known as lhs inside the function, and the right-hand operand, b, is known as rhs.

For member functions, the left-hand argument is represented by the this pointer, so sometimes I use the parameter name rhs by itself. You may have noticed this in the declarations for some Widget member functions on page 5. Which reminds me. I often use the Widget class in examples. "Widget" doesn't mean anything. It's just a name I sometimes use when I need an example class name. It has nothing to do with widgets in GUI toolkits.

I often name pointers following the rule that a pointer to an object of type T is called pt, "pointer to T." Here are some examples:

```
Widget *pw;                          // pw = ptr to Widget

class Airplane;
Airplane *pa;                        // pa = ptr to Airplane
```

```
class GameCharacter;
GameCharacter *pgc;                    // pgc = ptr to GameCharacter
```

I use a similar convention for references: rw might be a reference to a Widget and ra a reference to an Airplane.

I occasionally use the name mf when I'm talking about member functions.

Threading Considerations

As a language, C++ has no notion of threads — no notion of concurrency of any kind, in fact. Ditto for C++'s standard library. As far as C++ is concerned, multithreaded programs don't exist.

And yet they do. My focus in this book is on standard, portable C++, but I can't ignore the fact that thread safety is an issue many programmers confront. My approach to dealing with this chasm between standard C++ and reality is to point out places where the C++ constructs I examine are likely to cause problems in a threaded environment. That doesn't make this a book on multithreaded programming with C++. Far from it. Rather, it makes it a book on C++ programming that, while largely limiting itself to single-threaded considerations, acknowledges the existence of multithreading and tries to point out places where thread-aware programmers need to take particular care in evaluating the advice I offer.

If you're unfamiliar with multithreading or have no need to worry about it, you can ignore my threading-related remarks. If you are programming a threaded application or library, however, remember that my comments are little more than a starting point for the issues you'll need to address when using C++.

TR1 and Boost

You'll find references to TR1 and Boost throughout this book. Each has an Item that describes it in some detail (Item 54 for TR1, Item 55 for Boost), but, unfortunately, these Items are at the end of the book. (They're there because it works better that way. Really. I tried them in a number of other places.) If you like, you can turn to those Items and read them now, but if you'd prefer to start the book at the beginning instead of the end, the following executive summary will tide you over:

- TR1 ("Technical Report 1") is a specification for new functionality being added to C++'s standard library. This functionality takes the form of new class and function templates for things like hash ta-

bles, reference-counting smart pointers, regular expressions, and more. All TR1 components are in the namespace tr1 that's nested inside the namespace std.

- Boost is an organization and a web site (http://boost.org) offering portable, peer-reviewed, open source C++ libraries. Most TR1 functionality is based on work done at Boost, and until compiler vendors include TR1 in their C++ library distributions, the Boost web site is likely to remain the first stop for developers looking for TR1 implementations. Boost offers more than is available in TR1, however, so it's worth knowing about in any case.

1 Accustoming Yourself to C++

Regardless of your programming background, C++ is likely to take a little getting used to. It's a powerful language with an enormous range of features, but before you can harness that power and make effective use of those features, you have to accustom yourself to C++'s way of doing things. This entire book is about how to do that, but some things are more fundamental than others, and this chapter is about some of the most fundamental things of all.

Item 1: View C++ as a federation of languages.

In the beginning, C++ was just C with some object-oriented features tacked on. Even C++'s original name, "C with Classes," reflected this simple heritage.

As the language matured, it grew bolder and more adventurous, adopting ideas, features, and programming strategies different from those of C with Classes. Exceptions required different approaches to structuring functions (see Item 29). Templates gave rise to new ways of thinking about design (see Item 41), and the STL defined an approach to extensibility unlike any most people had ever seen.

Today's C++ is a *multiparadigm programming language*, one supporting a combination of procedural, object-oriented, functional, generic, and metaprogramming features. This power and flexibility make C++ a tool without equal, but can also cause some confusion. All the "proper usage" rules seem to have exceptions. How are we to make sense of such a language?

The easiest way is to view C++ not as a single language but as a federation of related languages. Within a particular sublanguage, the rules tend to be simple, straightforward, and easy to remember. When you move from one sublanguage to another, however, the rules may

change. To make sense of C++, you have to recognize its primary sublanguages. Fortunately, there are only four:

- **C.** Way down deep, C++ is still based on C. Blocks, statements, the preprocessor, built-in data types, arrays, pointers, etc., all come from C. In many cases, C++ offers approaches to problems that are superior to their C counterparts (e.g., see Items 2 (alternatives to the preprocessor) and 13 (using objects to manage resources)), but when you find yourself working with the C part of C++, the rules for effective programming reflect C's more limited scope: no templates, no exceptions, no overloading, etc.
- **Object-Oriented C++.** This part of C++ is what C with Classes was all about: classes (including constructors and destructors), encapsulation, inheritance, polymorphism, virtual functions (dynamic binding), etc. This is the part of C++ to which the classic rules for object-oriented design most directly apply.
- **Template C++.** This is the generic programming part of C++, the one that most programmers have the least experience with. Template considerations pervade C++, and it's not uncommon for rules of good programming to include special template-only clauses (e.g., see Item 46 on facilitating type conversions in calls to template functions). In fact, templates are so powerful, they give rise to a completely new programming paradigm, *template metaprogramming* (TMP). Item 48 provides an overview of TMP, but unless you're a hard-core template junkie, you need not worry about it. The rules for TMP rarely interact with mainstream C++ programming.
- **The STL.** The STL is a template library, of course, but it's a very special template library. Its conventions regarding containers, iterators, algorithms, and function objects mesh beautifully, but templates and libraries can be built around other ideas, too. The STL has particular ways of doing things, and when you're working with the STL, you need to be sure to follow its conventions.

Keep these four sublanguages in mind, and don't be surprised when you encounter situations where effective programming requires that you change strategy when you switch from one sublanguage to another. For example, pass-by-value is generally more efficient than pass-by-reference for built-in (i.e., C-like) types, but when you move from the C part of C++ to Object-Oriented C++, the existence of user-defined constructors and destructors means that pass-by-reference-to-const is usually better. This is especially the case when working in Template C++, because there, you don't even know the type of object

you're dealing with. When you cross into the STL, however, you know that iterators and function objects are modeled on pointers in C, so for iterators and function objects in the STL, the old C pass-by-value rule applies again. (For all the details on choosing among parameter-passing options, see Item 20.)

C++, then, isn't a unified language with a single set of rules; it's a federation of four sublanguages, each with its own conventions. Keep these sublanguages in mind, and you'll find that C++ is a lot easier to understand.

Things to Remember

- Rules for effective C++ programming vary, depending on the part of C++ you are using.

Item 2: Prefer consts, enums, and inlines to #defines.

This Item might better be called "prefer the compiler to the preprocessor," because #define may be treated as if it's not part of the language *per se*. That's one of its problems. When you do something like this,

```
#define ASPECT_RATIO 1.653
```

the symbolic name ASPECT_RATIO may never be seen by compilers; it may be removed by the preprocessor before the source code ever gets to a compiler. As a result, the name ASPECT_RATIO may not get entered into the symbol table. This can be confusing if you get an error during compilation involving the use of the constant, because the error message may refer to 1.653, not ASPECT_RATIO. If ASPECT_RATIO were defined in a header file you didn't write, you'd have no idea where that 1.653 came from, and you'd waste time tracking it down. This problem can also crop up in a symbolic debugger, because, again, the name you're programming with may not be in the symbol table.

The solution is to replace the macro with a constant:

```
const double AspectRatio = 1.653;      // uppercase names are usually for
                                       // macros, hence the name change
```

As a language constant, AspectRatio is definitely seen by compilers and is certainly entered into their symbol tables. In addition, in the case of a floating point constant (such as in this example), use of the constant may yield smaller code than using a #define. That's because the preprocessor's blind substitution of the macro name ASPECT_RATIO with 1.653 could result in multiple copies of 1.653 in your object code, while the use of the constant AspectRatio should never result in more than one copy.

When replacing #defines with constants, two special cases are worth mentioning. The first is defining constant pointers. Because constant definitions are typically put in header files (where many different source files will include them), it's important that the *pointer* be declared const, usually in addition to what the pointer points to. To define a constant char*-based string in a header file, for example, you have to write const *twice*:

```
const char * const authorName = "Scott Meyers";
```

For a complete discussion of the meanings and uses of const, especially in conjunction with pointers, see Item 3. However, it's worth reminding you here that string objects are generally preferable to their char*-based progenitors, so authorName is often better defined this way:

```
const std::string authorName("Scott Meyers");
```

The second special case concerns class-specific constants. To limit the scope of a constant to a class, you must make it a member, and to ensure there's at most one copy of the constant, you must make it a *static* member:

```
class GamePlayer {
private:
  static const int NumTurns = 5;          // constant declaration
  int scores[NumTurns];                    // use of constant
  ...
};
```

What you see above is a *declaration* for NumTurns, not a definition. Usually, C++ requires that you provide a definition for anything you use, but class-specific constants that are static and of integral type (e.g., integers, chars, bools) are an exception. As long as you don't take their address, you can declare them and use them without providing a definition. If you do take the address of a class constant, or if your compiler incorrectly insists on a definition even if you don't take the address, you provide a separate definition like this:

```
const int GamePlayer::NumTurns;          // definition of NumTurns; see
                                         // below for why no value is given
```

You put this in an implementation file, not a header file. Because the initial value of class constants is provided where the constant is declared (e.g., NumTurns is initialized to 5 when it is declared), no initial value is permitted at the point of definition.

Note, by the way, that there's no way to create a class-specific constant using a #define, because #defines don't respect scope. Once a macro is defined, it's in force for the rest of the compilation (unless it's

#undefed somewhere along the line). Which means that not only can't #defines be used for class-specific constants, they also can't be used to provide any kind of encapsulation, i.e., there is no such thing as a "private" #define. Of course, const data members can be encapsulated; NumTurns is.

Older compilers may not accept the syntax above, because it used to be illegal to provide an initial value for a static class member at its point of declaration. Furthermore, in-class initialization is allowed only for integral types and only for constants. In cases where the above syntax can't be used, you put the initial value at the point of definition:

```
class CostEstimate {
private:
  static const double FudgeFactor;          // declaration of static class
  ...                                       // constant; goes in header file
};

const double                                // definition of static class
  CostEstimate::FudgeFactor = 1.35;         // constant; goes in impl. file
```

This is all you need almost all the time. The only exception is when you need the value of a class constant during compilation of the class, such as in the declaration of the array GamePlayer::scores above (where compilers insist on knowing the size of the array during compilation). Then the accepted way to compensate for compilers that (incorrectly) forbid the in-class specification of initial values for static integral class constants is to use what is affectionately (and non-pejoratively) known as "the enum hack." This technique takes advantage of the fact that the values of an enumerated type can be used where ints are expected, so GamePlayer could just as well be defined like this:

```
class GamePlayer {
private:
  enum { NumTurns = 5 };          // "the enum hack" — makes
                                  // NumTurns a symbolic name for 5
  int scores[NumTurns];           // fine

  ...
};
```

The enum hack is worth knowing about for several reasons. First, the enum hack behaves in some ways more like a #define than a const does, and sometimes that's what you want. For example, it's legal to take the address of a const, but it's not legal to take the address of an enum, and it's typically not legal to take the address of a #define, either. If you don't want to let people get a pointer or reference to one

of your integral constants, an enum is a good way to enforce that constraint. (For more on enforcing design constraints through coding decisions, consult Item 18.) Also, though good compilers won't set aside storage for const objects of integral types (unless you create a pointer or reference to the object), sloppy compilers may, and you may not be willing to set aside memory for such objects. Like #defines, enums never result in that kind of unnecessary memory allocation.

A second reason to know about the enum hack is purely pragmatic. Lots of code employs it, so you need to recognize it when you see it. In fact, the enum hack is a fundamental technique of template metaprogramming (see Item 48).

Getting back to the preprocessor, another common (mis)use of the #define directive is using it to implement macros that look like functions but that don't incur the overhead of a function call. Here's a macro that calls some function f with the greater of the macro's arguments:

```
// call f with the maximum of a and b
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))
```

Macros like this have so many drawbacks, just thinking about them is painful.

Whenever you write this kind of macro, you have to remember to parenthesize all the arguments in the macro body. Otherwise you can run into trouble when somebody calls the macro with an expression. But even if you get that right, look at the weird things that can happen:

```
int a = 5, b = 0;

CALL_WITH_MAX(++a, b);          // a is incremented twice
CALL_WITH_MAX(++a, b+10);       // a is incremented once
```

Here, the number of times that a is incremented before calling f depends on what it is being compared with!

Fortunately, you don't need to put up with this nonsense. You can get all the efficiency of a macro plus all the predictable behavior and type safety of a regular function by using a template for an inline function (see Item 30):

```
template<typename T>                            // because we don't
inline void callWithMax(const T& a, const T& b) // know what T is, we
{                                               // pass by reference-to-
  f(a > b ? a : b);                             // const — see Item 20
}
```

This template generates a whole family of functions, each of which takes two objects of the same type and calls f with the greater of the

two objects. There's no need to parenthesize parameters inside the function body, no need to worry about evaluating parameters multiple times, etc. Furthermore, because callWithMax is a real function, it obeys scope and access rules. For example, it makes perfect sense to talk about an inline function that is private to a class. In general, there's just no way to do that with a macro.

Given the availability of consts, enums, and inlines, your need for the preprocessor (especially #define) is reduced, but it's not eliminated. #include remains essential, and #ifdef/#ifndef continue to play important roles in controlling compilation. It's not yet time to retire the preprocessor, but you should definitely give it long and frequent vacations.

Things to Remember

- For simple constants, prefer const objects or enums to #defines.
- For function-like macros, prefer inline functions to #defines.

Item 3: Use const whenever possible.

The wonderful thing about const is that it allows you to specify a semantic constraint — a particular object should *not* be modified — and compilers will enforce that constraint. It allows you to communicate to both compilers and other programmers that a value should remain invariant. Whenever that is true, you should be sure to say so, because that way you enlist your compilers' aid in making sure the constraint isn't violated.

The const keyword is remarkably versatile. Outside of classes, you can use it for constants at global or namespace scope (see Item 2), as well as for objects declared static at file, function, or block scope. Inside classes, you can use it for both static and non-static data members. For pointers, you can specify whether the pointer itself is const, the data it points to is const, both, or neither:

```
char greeting[] = "Hello";
char *p = greeting;                    // non-const pointer,
                                       // non-const data
const char *p = greeting;              // non-const pointer,
                                       // const data
char * const p = greeting;             // const pointer,
                                       // non-const data
const char * const p = greeting;       // const pointer,
                                       // const data
```

This syntax isn't as capricious as it may seem. If the word const appears to the left of the asterisk, what's *pointed to* is constant; if the word const appears to the right of the asterisk, the *pointer itself* is constant; if const appears on both sides, both are constant.†

When what's pointed to is constant, some programmers list const before the type. Others list it after the type but before the asterisk. There is no difference in meaning, so the following functions take the same parameter type:

```
void f1(const Widget *pw);          // f1 takes a pointer to a
                                    // constant Widget object
void f2(Widget const *pw);          // so does f2
```

Because both forms exist in real code, you should accustom yourself to both of them.

STL iterators are modeled on pointers, so an iterator acts much like a T* pointer. Declaring an iterator const is like declaring a pointer const (i.e., declaring a T* const pointer): the iterator isn't allowed to point to something different, but the thing it points to may be modified. If you want an iterator that points to something that can't be modified (i.e., the STL analogue of a const T* pointer), you want a const_iterator:

```
std::vector<int> vec;
...
const std::vector<int>::iterator iter =     // iter acts like a T* const
  vec.begin();
*iter = 10;                                 // OK, changes what iter points to
++iter;                                     // error! iter is const

std::vector<int>::const_iterator cIter =    //cIter acts like a const T*
  vec.begin();
*cIter = 10;                                // error! *cIter is const
++cIter;                                    // fine, changes cIter
```

Some of the most powerful uses of const stem from its application to function declarations. Within a function declaration, const can refer to the function's return value, to individual parameters, and, for member functions, to the function as a whole.

Having a function return a constant value often makes it possible to reduce the incidence of client errors without giving up safety or efficiency. For example, consider the declaration of the operator* function for rational numbers that is explored in Item 24:

```
class Rational { ... };
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

† Some people find it helpful to read pointer declarations right to left, e.g., to read const char * const p as "p is a constant pointer to constant chars."

Many programmers squint when they first see this. Why should the result of operator* be a const object? Because if it weren't, clients would be able to commit atrocities like this:

```
Rational a, b, c;
...
(a * b) = c;                    // invoke operator= on the
                                // result of a*b!
```

I don't know why any programmer would want to make an assignment to the product of two numbers, but I do know that many programmers have tried to do it without wanting to. All it takes is a simple typo (and a type that can be implicitly converted to bool):

```
if (a * b = c) ...              // oops, meant to do a comparison!
```

Such code would be flat-out illegal if a and b were of a built-in type. One of the hallmarks of good user-defined types is that they avoid gratuitous incompatibilities with the built-ins (see also Item 18), and allowing assignments to the product of two numbers seems pretty gratuitous to me. Declaring operator*'s return value const prevents it, and that's why it's The Right Thing To Do.

There's nothing particularly new about const parameters — they act just like local const objects, and you should use both whenever you can. Unless you need to be able to modify a parameter or local object, be sure to declare it const. It costs you only the effort to type six characters, and it can save you from annoying errors such as the "I meant to type '==' but I accidently typed '='" mistake we just saw.

const Member Functions

The purpose of const on member functions is to identify which member functions may be invoked on const objects. Such member functions are important for two reasons. First, they make the interface of a class easier to understand. It's important to know which functions may modify an object and which may not. Second, they make it possible to work with const objects. That's a critical aspect of writing efficient code, because, as Item 20 explains, one of the fundamental ways to improve a C++ program's performance is to pass objects by reference-to-const. That technique is viable only if there are const member functions with which to manipulate the resulting const-qualified objects.

Many people overlook the fact that member functions differing *only* in their constness can be overloaded, but this is an important feature of C++. Consider a class for representing a block of text:

```
class TextBlock {
public:
  ...
  const char& operator[](std::size_t position) const    // operator[] for
  { return text[position]; }                            // const objects
  char& operator[](std::size_t position)                // operator[] for
  { return text[position]; }                            // non-const objects
private:
   std::string text;
};
```

TextBlock's operator[]s can be used like this:

```
TextBlock tb("Hello");
std::cout << tb[0];                     // calls non-const
                                        // TextBlock::operator[]
const TextBlock ctb("World");
std::cout << ctb[0];                    // calls const TextBlock::operator[]
```

Incidentally, const objects most often arise in real programs as a result of being passed by pointer- or reference-to-const. The example of ctb above is artificial. This is more realistic:

```
void print(const TextBlock& ctb)        // in this function, ctb is const
{
  std::cout << ctb[0];                  // calls const TextBlock::operator[]
  ...
}
```

By overloading operator[] and giving the different versions different return types, you can have const and non-const TextBlocks handled differently:

```
std::cout << tb[0];                     // fine — reading a
                                        // non-const TextBlock

tb[0] = 'x';                            // fine — writing a
                                        // non-const TextBlock

std::cout << ctb[0];                    // fine — reading a
                                        // const TextBlock

ctb[0] = 'x';                           // error! — writing a
                                        // const TextBlock
```

Note that the error here has only to do with the *return type* of the operator[] that is called; the calls to operator[] themselves are all fine. The error arises out of an attempt to make an assignment to a const char&, because that's the return type from the const version of operator[].

Also note that the return type of the non-const operator[] is a *reference* to a char — a char itself would not do. If operator[] did return a simple char, statements like this wouldn't compile:

```
tb[0] = 'x';
```

That's because it's never legal to modify the return value of a function that returns a built-in type. Even if it were legal, the fact that C++ returns objects by value (see Item 20) would mean that a *copy* of tb.text[0] would be modified, not tb.text[0] itself, and that's not the behavior you want.

Let's take a brief time-out for philosophy. What does it mean for a member function to be const? There are two prevailing notions: *bitwise constness* (also known as *physical constness*) and *logical constness*.

The bitwise const camp believes that a member function is const if and only if it doesn't modify any of the object's data members (excluding those that are static), i.e., if it doesn't modify any of the bits inside the object. The nice thing about bitwise constness is that it's easy to detect violations: compilers just look for assignments to data members. In fact, bitwise constness is C++'s definition of constness, and a const member function isn't allowed to modify any of the non-static data members of the object on which it is invoked.

Unfortunately, many member functions that don't act very const pass the bitwise test. In particular, a member function that modifies what a pointer *points to* frequently doesn't act const. But if only the *pointer* is in the object, the function is bitwise const, and compilers won't complain. That can lead to counterintuitive behavior. For example, suppose we have a TextBlock-like class that stores its data as a char* instead of a string, because it needs to communicate through a C API that doesn't understand string objects.

```
class CTextBlock {
public:
  ...
  char& operator[](std::size_t position) const    // inappropriate (but bitwise
  { return pText[position]; }                     // const) declaration of
                                                  // operator[]
private:
  char *pText;
};
```

This class (inappropriately) declares operator[] as a const member function, even though that function returns a reference to the object's internal data (a topic treated in depth in Item 28). Set that aside and

note that operator[]'s implementation doesn't modify pText in any way. As a result, compilers will happily generate code for operator[]; it is, after all, bitwise const, and that's all compilers check for. But look what it allows to happen:

```
const CTextBlock cctb("Hello");          // declare constant object
char *pc = &cctb[0];                     // call the const operator[] to get a
                                         // pointer to cctb's data
*pc = 'J';                               // cctb now has the value "Jello"
```

Surely there is something wrong when you create a constant object with a particular value and you invoke only const member functions on it, yet you still change its value!

This leads to the notion of logical constness. Adherents to this philosophy — and you should be among them — argue that a const member function might modify some of the bits in the object on which it's invoked, but only in ways that clients cannot detect. For example, your CTextBlock class might want to cache the length of the textblock whenever it's requested:

```
class CTextBlock {
public:
  ...
  std::size_t length() const;

private:
  char *pText;
  std::size_t textLength;                // last calculated length of textblock
  bool lengthIsValid;                    // whether length is currently valid
};

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);     // error! can't assign to textLength
    lengthIsValid = true;                // and lengthIsValid in a const
  }                                      // member function

  return textLength;
}
```

This implementation of length is certainly not bitwise const — both textLength and lengthIsValid may be modified — yet it seems as though it should be valid for const CTextBlock objects. Compilers disagree. They insist on bitwise constness. What to do?

The solution is simple: take advantage of C++'s const-related wiggle room known as mutable. mutable frees non-static data members from the constraints of bitwise constness:

```
class CTextBlock {
public:
  ...
  std::size_t length() const;
private:
  char *pText;
  mutable std::size_t textLength;          // these data members may
  mutable bool lengthIsValid;              // always be modified, even in
};                                         // const member functions

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);       // now fine
    lengthIsValid = true;                  // also fine
  }
  return textLength;
}
```

Avoiding Duplication in const and Non-const Member Functions

mutable is a nice solution to the bitwise-constness-is-not-what-I-had-in-mind problem, but it doesn't solve all const-related difficulties. For example, suppose that operator[] in TextBlock (and CTextBlock) not only returned a reference to the appropriate character, it also performed bounds checking, logged access information, maybe even did data integrity validation. Putting all this in both the const and the non-const operator[] functions (and not fretting that we now have implicitly inline functions of nontrivial length — see Item 30) yields this kind of monstrosity:

```
class TextBlock {
public:
  ...
  const char& operator[](std::size_t position) const
  {
    ...                                    // do bounds checking
    ...                                    // log access data
    ...                                    // verify data integrity
    return text[position];
  }
  char& operator[](std::size_t position)
  {
    ...                                    // do bounds checking
    ...                                    // log access data
    ...                                    // verify data integrity
    return text[position];
  }
private:
  std::string text;
};
```

Ouch! Can you say code duplication, along with its attendant compilation time, maintenance, and code-bloat headaches? Sure, it's possible to move all the code for bounds checking, etc. into a separate member function (private, naturally) that both versions of operator[] call, but you've still got the duplicated calls to that function and you've still got the duplicated return statement code.

What you really want to do is implement operator[] functionality once and use it twice. That is, you want to have one version of operator[] call the other one. And that brings us to casting away constness.

As a general rule, casting is such a bad idea, I've devoted an entire Item to telling you not to do it (Item 27), but code duplication is no picnic, either. In this case, the const version of operator[] does exactly what the non-const version does, it just has a const-qualified return type. Casting away the const on the return value is safe, in this case, because whoever called the non-const operator[] must have had a non-const object in the first place. Otherwise they couldn't have called a non-const function. So having the non-const operator[] call the const version is a safe way to avoid code duplication, even though it requires a cast. Here's the code, but it may be clearer after you read the explanation that follows:

```
class TextBlock {
public:
  ...
  const char& operator[](std::size_t position) const         // same as before
  {
    ...
    ...
    ...
    return text[position];
  }
  char& operator[](std::size_t position)              // now just calls const op[]
  {
    return
      const_cast<char&>(                             // cast away const on
                                                     // op[]'s return type;
        static_cast<const TextBlock&>(*this)         // add const to *this's type;
          [position]                                 // call const version of op[]
      );
  }
...
};
```

As you can see, the code has two casts, not one. We want the non-const operator[] to call the const one, but if, inside the non-const operator[], we just call operator[], we'll recursively call ourselves. That's only entertaining the first million or so times. To avoid infinite recursion, we have to specify that we want to call the const operator[], but there's no direct way to do that. Instead, we cast *this from its native type of TextBlock& to const TextBlock&. Yes, we use a cast to *add* const! So we have two casts: one to add const to *this (so that our call to operator[] will call the const version), the second to remove the const from the const operator[]'s return value.

The cast that adds const is just forcing a safe conversion (from a non-const object to a const one), so we use a static_cast for that. The one that removes const can be accomplished only via a const_cast, so we don't really have a choice there. (Technically, we do. A C-style cast would also work, but, as I explain in Item 27, such casts are rarely the right choice. If you're unfamiliar with static_cast or const_cast, Item 27 contains an overview.)

On top of everything else, we're calling an operator in this example, so the syntax is a little strange. The result may not win any beauty contests, but it has the desired effect of avoiding code duplication by implementing the non-const version of operator[] in terms of the const version. Whether achieving that goal is worth the ungainly syntax is something only you can determine, but the technique of implementing a non-const member function in terms of its const twin is definitely worth knowing.

Even more worth knowing is that trying to do things the other way around — avoiding duplication by having the const version call the non-const version — is *not* something you want to do. Remember, a const member function promises never to change the logical state of its object, but a non-const member function makes no such promise. If you were to call a non-const function from a const one, you'd run the risk that the object you'd promised not to modify would be changed. That's why having a const member function call a non-const one is wrong: the object could be changed. In fact, to get the code to compile, you'd have to use a const_cast to get rid of the const on *this, a clear sign of trouble. The reverse calling sequence — the one we used above — is safe: the non-const member function can do whatever it wants with an object, so calling a const member function imposes no risk. That's why a static_cast works on *this in that case: there's no const-related danger.

As I noted at the beginning of this Item, const is a wonderful thing. On pointers and iterators; on the objects referred to by pointers, iterators,

and references; on function parameters and return types; on local variables; and on member functions, const is a powerful ally. Use it whenever you can. You'll be glad you did.

Things to Remember

- Declaring something const helps compilers detect usage errors. const can be applied to objects at any scope, to function parameters and return types, and to member functions as a whole.
- Compilers enforce bitwise constness, but you should program using logical constness.
- When const and non-const member functions have essentially identical implementations, code duplication can be avoided by having the non-const version call the const version.

Item 4: Make sure that objects are initialized before they're used.

C++ can seem rather fickle about initializing the values of objects. For example, if you say this,

```
int x;
```

in some contexts, x is guaranteed to be initialized (to zero), but in others, it's not. If you say this,

```
class Point {
  int x, y;
};

...

Point p;
```

p's data members are sometimes guaranteed to be initialized (to zero), but sometimes they're not. If you're coming from a language where uninitialized objects can't exist, pay attention, because this is important.

Reading uninitialized values yields undefined behavior. On some platforms, the mere act of reading an uninitialized value can halt your program. More typically, the result of the read will be semi-random bits, which will then pollute the object you read the bits into, eventually leading to inscrutable program behavior and a lot of unpleasant debugging.

Now, there are rules that describe when object initialization is guaranteed to take place and when it isn't. Unfortunately, the rules are com-

plicated — too complicated to be worth memorizing, in my opinion. In general, if you're in the C part of C++ (see Item 1) and initialization would probably incur a runtime cost, it's not guaranteed to take place. If you cross into the non-C parts of C++, things sometimes change. This explains why an array (from the C part of C++) isn't necessarily guaranteed to have its contents initialized, but a vector (from the STL part of C++) is.

The best way to deal with this seemingly indeterminate state of affairs is to *always* initialize your objects before you use them. For non-member objects of built-in types, you'll need to do this manually. For example:

```
int x = 0;                                  // manual initialization of an int

const char * text = "A C-style string";     // manual initialization of a
                                            // pointer (see also Item 3)

double d;                                   // "initialization" by reading from
std::cin >> d;                              // an input stream
```

For almost everything else, the responsibility for initialization falls on constructors. The rule there is simple: make sure that all constructors initialize everything in the object.

The rule is easy to follow, but it's important not to confuse assignment with initialization. Consider a constructor for a class representing entries in an address book:

```
class PhoneNumber { ... };

class ABEntry {                             // ABEntry = "Address Book Entry"
public:
  ABEntry(const std::string& name, const std::string& address,
          const std::list<PhoneNumber>& phones);
private:
  std::string theName;
  std::string theAddress;
  std::list<PhoneNumber> thePhones;
  int numTimesConsulted;
};

ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
{
  theName = name;                           // these are all assignments,
  theAddress = address;                     // not initializations
  thePhones = phones;
  numTimesConsulted = 0;
}
```

This will yield ABEntry objects with the values you expect, but it's still not the best approach. The rules of C++ stipulate that data members of an object are initialized *before* the body of a constructor is entered. Inside the ABEntry constructor, theName, theAddress, and thePhones aren't being initialized, they're being *assigned*. Initialization took place earlier — when their default constructors were automatically called prior to entering the body of the ABEntry constructor. This isn't true for numTimesConsulted, because it's a built-in type. For it, there's no guarantee it was initialized at all prior to its assignment.

A better way to write the ABEntry constructor is to use the member initialization list instead of assignments:

```
ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
: theName(name),
  theAddress(address),                  // these are now all initializations
  thePhones(phones),
  numTimesConsulted(0)
{}                                      // the ctor body is now empty
```

This constructor yields the same end result as the one above, but it will often be more efficient. The assignment-based version first called default constructors to initialize theName, theAddress, and thePhones, then promptly assigned new values on top of the default-constructed ones. All the work performed in those default constructions was therefore wasted. The member initialization list approach avoids that problem, because the arguments in the initialization list are used as constructor arguments for the various data members. In this case, theName is copy-constructed from name, theAddress is copy-constructed from address, and thePhones is copy-constructed from phones. For most types, a single call to a copy constructor is more efficient — sometimes *much* more efficient — than a call to the default constructor followed by a call to the copy assignment operator.

For objects of built-in type like numTimesConsulted, there is no difference in cost between initialization and assignment, but for consistency, it's often best to initialize everything via member initialization. Similarly, you can use the member initialization list even when you want to default-construct a data member; just specify nothing as an initialization argument. For example, if ABEntry had a constructor taking no parameters, it could be implemented like this:

```
ABEntry::ABEntry()
: theName(),                            // call theName's default ctor;
  theAddress(),                         // do the same for theAddress;
  thePhones(),                          // and for thePhones;
  numTimesConsulted(0)                  // but explicitly initialize
{}                                      // numTimesConsulted to zero
```

Because compilers will automatically call default constructors for data members of user-defined types when those data members have no initializers on the member initialization list, some programmers consider the above approach overkill. That's understandable, but having a policy of always listing every data member on the initialization list avoids having to remember which data members may go uninitialized if they are omitted. Because numTimesConsulted is of a built-in type, for example, leaving it off a member initialization list could open the door to undefined behavior.

Sometimes the initialization list *must* be used, even for built-in types. For example, data members that are const or are references must be initialized; they can't be assigned (see also Item 5). To avoid having to memorize when data members must be initialized in the member initialization list and when it's optional, the easiest choice is to *always* use the initialization list. It's sometimes required, and it's often more efficient than assignments.

Many classes have multiple constructors, and each constructor has its own member initialization list. If there are many data members and/or base classes, the existence of multiple initialization lists introduces undesirable repetition (in the lists) and boredom (in the programmers). In such cases, it's not unreasonable to omit entries in the lists for data members where assignment works as well as true initialization, moving the assignments to a single (typically private) function that all the constructors call. This approach can be especially helpful if the true initial values for the data members are to be read from a file or looked up in a database. In general, however, true member initialization (via an initialization list) is preferable to pseudo-initialization via assignment.

One aspect of C++ that isn't fickle is the order in which an object's data is initialized. This order is always the same: base classes are initialized before derived classes (see also Item 12), and within a class, data members are initialized in the order in which they are declared. In ABEntry, for example, theName will always be initialized first, theAddress second, thePhones third, and numTimesConsulted last. This is true even if they are listed in a different order on the member initialization list (something that's unfortunately legal). To avoid reader confusion, as well as the possibility of some truly obscure behavioral bugs, always list members in the initialization list in the same order as they're declared in the class.

Once you've taken care of explicitly initializing non-member objects of built-in types and you've ensured that your constructors initialize their base classes and data members using the member initialization

list, there's only one more thing to worry about. That thing is — take a deep breath — the order of initialization of non-local static objects defined in different translation units.

Let's pick that phrase apart bit by bit.

A *static object* is one that exists from the time it's constructed until the end of the program. Stack and heap-based objects are thus excluded. Included are global objects, objects defined at namespace scope, objects declared static inside classes, objects declared static inside functions, and objects declared static at file scope. Static objects inside functions are known as *local static objects* (because they're local to a function), and the other kinds of static objects are known as *non-local static objects*. Static objects are destroyed when the program exits, i.e., their destructors are called when main finishes executing.

A *translation unit* is the source code giving rise to a single object file. It's basically a single source file, plus all of its #include files.

The problem we're concerned with, then, involves at least two separately compiled source files, each of which contains at least one non-local static object (i.e., an object that's global, at namespace scope, or static in a class or at file scope). And the actual problem is this: if initialization of a non-local static object in one translation unit uses a non-local static object in a different translation unit, the object it uses could be uninitialized, because *the relative order of initialization of non-local static objects defined in different translation units is undefined*.

An example will help. Suppose you have a FileSystem class that makes files on the Internet look like they're local. Since your class makes the world look like a single file system, you might create a special object at global or namespace scope representing the single file system:

```
class FileSystem {                      // from your library's header file
public:
  ...
  std::size_t numDisks() const;         // one of many member functions
  ...
};

extern FileSystem tfs;                  // declare object for clients to use
                                        // ("tfs" = "the file system"); definition
                                        // is in some .cpp file in your library
```

A FileSystem object is decidedly non-trivial, so use of the tfs object before it has been constructed would be disastrous.

Now suppose some client creates a class for directories in a file system. Naturally, their class uses the tfs object:

```
class Directory {                          // created by library client
public:
   Directory( params );
   ...
};

Directory::Directory( params )
{
   ...
   std::size_t disks = tfs.numDisks();     // use the tfs object
   ...
}
```

Further suppose this client decides to create a single Directory object for temporary files:

```
Directory tempDir( params );               // directory for temporary files
```

Now the importance of initialization order becomes apparent: unless tfs is initialized before tempDir, tempDir's constructor will attempt to use tfs before it's been initialized. But tfs and tempDir were created by different people at different times in different source files — they're non-local static objects defined in different translation units. How can you be sure that tfs will be initialized before tempDir?

You can't. Again, *the relative order of initialization of non-local static objects defined in different translation units is undefined*. There is a reason for this. Determining the "proper" order in which to initialize non-local static objects is hard. Very hard. Unsolvably hard. In its most general form — with multiple translation units and non-local static objects generated through implicit template instantiations (which may themselves arise via implicit template instantiations) — it's not only impossible to determine the right order of initialization, it's typically not even worth looking for special cases where it *is* possible to determine the right order.

Fortunately, a small design change eliminates the problem entirely. All that has to be done is to move each non-local static object into its own function, where it's declared static. These functions return references to the objects they contain. Clients then call the functions instead of referring to the objects. In other words, non-local static objects are replaced with *local* static objects. (Aficionados of design patterns will recognize this as a common implementation of the Singleton pattern.†)

This approach is founded on C++'s guarantee that local static objects are initialized when the object's definition is first encountered during a call to that function. So if you replace direct accesses to non-local

† Actually, it's only *part* of a Singleton implementation. An essential part of Singleton I ignore in this Item is preventing the creation of multiple objects of a particular type.

static objects with calls to functions that return references to local static objects, you're guaranteed that the references you get back will refer to initialized objects. As a bonus, if you never call a function emulating a non-local static object, you never incur the cost of constructing and destructing the object, something that can't be said for true non-local static objects.

Here's the technique applied to both tfs and tempDir:

```
class FileSystem { ... };                // as before

FileSystem& tfs()                        // this replaces the tfs object; it could be
{                                        // static in the FileSystem class
  static FileSystem fs;                  // define and initialize a local static object
  return fs;                             // return a reference to it
}

class Directory { ... };                 // as before

Directory::Directory( params )           // as before, except references to tfs are
{                                        // now to tfs()
  ...
  std::size_t disks = tfs().numDisks();
  ...
}

Directory& tempDir()                     // this replaces the tempDir object; it
{                                        // could be static in the Directory class
  static Directory td( params );         // define/initialize local static object
  return td;                             // return reference to it
}
```

Clients of this modified system program exactly as they used to, except they now refer to tfs() and tempDir() instead of tfs and tempDir. That is, they use functions returning references to objects instead of using the objects themselves.

The reference-returning functions dictated by this scheme are always simple: define and initialize a local static object on line 1, return it on line 2. This simplicity makes them excellent candidates for inlining, especially if they're called frequently (see Item 30). On the other hand, the fact that these functions contain static objects makes them problematic in multithreaded systems. Then again, any kind of non-const static object — local or non-local — is trouble waiting to happen in the presence of multiple threads. One way to deal with such trouble is to manually invoke all the reference-returning functions during the single-threaded startup portion of the program. This eliminates initialization-related race conditions.

Of course, the idea of using reference-returning functions to prevent initialization order problems is dependent on there being a reasonable

initialization order for your objects in the first place. If you have a system where object A must be initialized before object B, but A's initialization is dependent on B's having already been initialized, you are going to have problems, and frankly, you deserve them. If you steer clear of such pathological scenarios, however, the approach described here should serve you nicely, at least in single-threaded applications.

To avoid using objects before they're initialized, then, you need to do only three things. First, manually initialize non-member objects of built-in types. Second, use member initialization lists to initialize all parts of an object. Finally, design around the initialization order uncertainty that afflicts non-local static objects defined in separate translation units.

Things to Remember

- Manually initialize objects of built-in type, because C++ only sometimes initializes them itself.
- In a constructor, prefer use of the member initialization list to assignment inside the body of the constructor. List data members in the initialization list in the same order they're declared in the class.
- Avoid initialization order problems across translation units by replacing non-local static objects with local static objects.

2 Constructors, Destructors, and Assignment Operators

Almost every class you write will have one or more constructors, a destructor, and a copy assignment operator. Little wonder. These are your bread-and-butter functions, the ones that control the fundamental operations of bringing a new object into existence and making sure it's initialized, getting rid of an object and making sure it's properly cleaned up, and giving an object a new value. Making mistakes in these functions will lead to far-reaching — and unpleasant — repercussions throughout your classes, so it's vital that you get them right. In this chapter, I offer guidance on putting together the functions that comprise the backbone of well-formed classes.

Item 5: Know what functions C++ silently writes and calls.

When is an empty class not an empty class? When C++ gets through with it. If you don't declare them yourself, compilers will declare their own versions of a copy constructor, a copy assignment operator, and a destructor. Furthermore, if you declare no constructors at all, compilers will also declare a default constructor for you. All these functions will be both public and inline (see Item 30). As a result, if you write

```
class Empty{};
```

it's essentially the same as if you'd written this:

```
class Empty {
public:
  Empty() { ... }                                  // default constructor
  Empty(const Empty& rhs) { ... }                  // copy constructor

  ~Empty() { ... }                                 // destructor — see below
                                                   // for whether it's virtual

  Empty& operator=(const Empty& rhs) { ... }       // copy assignment operator
};
```

These functions are generated only if they are needed, but it doesn't take much to need them. The following code will cause each function to be generated:

```
Empty e1;                          // default constructor;
                                   // destructor
Empty e2(e1);                      // copy constructor
e2 = e1;                           // copy assignment operator
```

Given that compilers are writing functions for you, what do the functions do? Well, the default constructor and the destructor primarily give compilers a place to put "behind the scenes" code such as invocation of constructors and destructors of base classes and non-static data members. Note that the generated destructor is non-virtual (see Item 7) unless it's for a class inheriting from a base class that itself declares a virtual destructor (in which case the function's virtualness comes from the base class).

As for the copy constructor and the copy assignment operator, the compiler-generated versions simply copy each non-static data member of the source object over to the target object. For example, consider a NamedObject template that allows you to associate names with objects of type T:

```
template<typename T>
class NamedObject {
public:
  NamedObject(const char *name, const T& value);
  NamedObject(const std::string& name, const T& value);

  ...

private:
  std::string nameValue;
  T objectValue;
};
```

Because a constructor is declared in NamedObject, compilers won't generate a default constructor. This is important. It means that if you've carefully engineered a class to require constructor arguments, you don't have to worry about compilers overriding your decision by blithely adding a constructor that takes no arguments.

NamedObject declares neither copy constructor nor copy assignment operator, so compilers will generate those functions (if they are needed). Look, then, at this use of the copy constructor:

```
NamedObject<int> no1("Smallest Prime Number", 2);
NamedObject<int> no2(no1);                  // calls copy constructor
```

The copy constructor generated by compilers must initialize no2.nameValue and no2.objectValue using no1.nameValue and no1.objectValue, respectively. The type of nameValue is string, and the standard string type has a copy constructor, so no2.nameValue will be initialized by calling the string copy constructor with no1.nameValue as its argument. On the other hand, the type of NamedObject<int>::objectValue is int (because T is int for this template instantiation), and int is a built-in type, so no2.objectValue will be initialized by copying the bits in no1.objectValue.

The compiler-generated copy assignment operator for NamedObject<int> would behave essentially the same way, but in general, compiler-generated copy assignment operators behave as I've described only when the resulting code is both legal and has a reasonable chance of making sense. If either of these tests fails, compilers will refuse to generate an operator= for your class.

For example, suppose NamedObject were defined like this, where nameValue is a *reference* to a string and objectValue is a *const* T:

```
template<typename T>
class NamedObject {
public:
  // this ctor no longer takes a const name, because nameValue
  // is now a reference-to-non-const string. The char* constructor
  // is gone, because we must have a string to refer to.
  NamedObject(std::string& name, const T& value);

  ...                                   // as above, assume no
                                        // operator= is declared
private:
  std::string& nameValue;               // this is now a reference
  const T objectValue;                  // this is now const
};
```

Now consider what should happen here:

```
std::string newDog("Persephone");
std::string oldDog("Satch");

NamedObject<int> p(newDog, 2);          // when I originally wrote this, our
                                        // dog Persephone was about to
                                        // have her second birthday

NamedObject<int> s(oldDog, 36);         // the family dog Satch (from my
                                        // childhood) would be 36 if she
                                        // were still alive

p = s;                                  // what should happen to
                                        // the data members in p?
```

Before the assignment, both p.nameValue and s.nameValue refer to string objects, though not the same ones. How should the assignment affect p.nameValue? After the assignment, should p.nameValue refer to the

string referred to by s.nameValue, i.e., should the reference itself be modified? If so, that breaks new ground, because C++ doesn't provide a way to make a reference refer to a different object. Alternatively, should the string object to which p.nameValue refers be modified, thus affecting other objects that hold pointers or references to that string, i.e., objects not directly involved in the assignment? Is that what the compiler-generated copy assignment operator should do?

Faced with this conundrum, C++ refuses to compile the code. If you want to support copy assignment in a class containing a reference member, you must define the copy assignment operator yourself. Compilers behave similarly for classes containing const members (such as objectValue in the modified class above). It's not legal to modify const members, so compilers are unsure how to treat them during an implicitly generated assignment function. Finally, compilers reject implicit copy assignment operators in derived classes that inherit from base classes declaring the copy assignment operator private. After all, compiler-generated copy assignment operators for derived classes are supposed to handle base class parts, too (see Item 12), but in doing so, they certainly can't invoke member functions the derived class has no right to call.

Things to Remember

- Compilers may implicitly generate a class's default constructor, copy constructor, copy assignment operator, and destructor.

Item 6: Explicitly disallow the use of compiler-generated functions you do not want.

Real estate agents sell houses, and a software system supporting such agents would naturally have a class representing homes for sale:

```
class HomeForSale { ... };
```

As every real estate agent will be quick to point out, every property is unique — no two are exactly alike. That being the case, the idea of making a *copy* of a HomeForSale object makes little sense. How can you copy something that's inherently unique? You'd thus like attempts to copy HomeForSale objects to not compile:

```
HomeForSale h1;
HomeForSale h2;
HomeForSale h3(h1);          // attempt to copy h1 — should
                             // not compile!
h1 = h2;                     // attempt to copy h2 — should
                             // not compile!
```

Alas, preventing such compilation isn't completely straightforward. Usually, if you don't want a class to support a particular kind of functionality, you simply don't declare the function that would provide it. This strategy doesn't work for the copy constructor and copy assignment operator, because, as Item 5 points out, if you don't declare them and somebody tries to call them, compilers declare them for you.

This puts you in a bind. If you don't declare a copy constructor or a copy assignment operator, compilers may generate them for you. Your class thus supports copying. If, on the other hand, you do declare these functions, your class still supports copying. But the goal here is to *prevent* copying!

The key to the solution is that all the compiler generated functions are public. To prevent these functions from being generated, you must declare them yourself, but there is nothing that requires that *you* declare them public. Instead, declare the copy constructor and the copy assignment operator *private*. By declaring a member function explicitly, you prevent compilers from generating their own version, and by making the function private, you keep people from calling it.

Mostly. The scheme isn't foolproof, because member and friend functions can still call your private functions. *Unless*, that is, you are clever enough not to *define* them. Then if somebody inadvertently calls one, they'll get an error at link-time. This trick — declaring member functions private and deliberately not implementing them — is so well established, it's used to prevent copying in several classes in C++'s iostreams library. Take a look, for example, at the definitions of ios_base, basic_ios, and sentry in your standard library implementation. You'll find that in each case, both the copy constructor and the copy assignment operator are declared private and are not defined.

Applying the trick to HomeForSale is easy:

```
class HomeForSale {
public:
  ...
private:
  ...
  HomeForSale(const HomeForSale&);                    // declarations only
  HomeForSale& operator=(const HomeForSale&);
};
```

You'll note that I've omitted the names of the functions' parameters. This isn't required, it's just a common convention. After all, the functions will never be implemented, much less used, so what's the point in specifying parameter names?

With the above class definition, compilers will thwart client attempts to copy HomeForSale objects, and if you inadvertently try to do it in a

member or a friend function, the linker will complain.

It's possible to move the link-time error up to compile time (always a good thing — earlier error detection is better than later) by declaring the copy constructor and copy assignment operator private not in HomeForSale itself, but in a base class specifically designed to prevent copying. The base class is simplicity itself:

```
class Uncopyable {
protected:                                          // allow construction
  Uncopyable() {}                                   // and destruction of
  ~Uncopyable() {}                                  // derived objects...

private:
  Uncopyable(const Uncopyable&);                    // ...but prevent copying
  Uncopyable& operator=(const Uncopyable&);
};
```

To keep HomeForSale objects from being copied, all we have to do now is inherit from Uncopyable:

```
class HomeForSale: private Uncopyable {             // class no longer
  ...                                               // declares copy ctor or
};                                                  // copy assign. operator
```

This works, because compilers will try to generate a copy constructor and a copy assignment operator if anybody — even a member or friend function — tries to copy a HomeForSale object. As Item 12 explains, the compiler-generated versions of these functions will try to call their base class counterparts, and those calls will be rejected, because the copying operations are private in the base class.

The implementation and use of Uncopyable include some subtleties, such as the fact that inheritance from Uncopyable needn't be public (see Items 32 and 39) and that Uncopyable's destructor need not be virtual (see Item 7). Because Uncopyable contains no data, it's eligible for the empty base class optimization described in Item 39, but because it's a base class, use of this technique could lead to multiple inheritance (see Item 40). Multiple inheritance, in turn, can sometimes disable the empty base class optimization (again, see Item 39). In general, you can ignore these subtleties and just use Uncopyable as shown, because it works precisely as advertised. You can also use the version available at Boost (see Item 55). That class is named noncopyable. It's a fine class, I just find the name a bit un-, er, *non*natural.

Things to Remember

- To disallow functionality automatically provided by compilers, declare the corresponding member functions private and give no implementations. Using a base class like Uncopyable is one way to do this.

Item 7: Declare destructors virtual in polymorphic base classes.

There are lots of ways to keep track of time, so it would be reasonable to create a TimeKeeper base class along with derived classes for different approaches to timekeeping:

```
class TimeKeeper {
public:
  TimeKeeper();
  ~TimeKeeper();
  ...
};

class AtomicClock: public TimeKeeper { ... };

class WaterClock: public TimeKeeper { ... };

class WristWatch: public TimeKeeper { ... };
```

Many clients will want access to the time without worrying about the details of how it's calculated, so a *factory function* — a function that returns a base class pointer to a newly-created derived class object — can be used to return a pointer to a timekeeping object:

```
TimeKeeper* getTimeKeeper();          // returns a pointer to a dynamic-
                                      // ally allocated object of a class
                                      // derived from TimeKeeper
```

In keeping with the conventions of factory functions, the objects returned by getTimeKeeper are on the heap, so to avoid leaking memory and other resources, it's important that each returned object be properly deleted:

```
TimeKeeper *ptk = getTimeKeeper();    // get dynamically allocated object
                                      // from TimeKeeper hierarchy
...                                   // use it
delete ptk;                           // release it to avoid resource leak
```

Item 13 explains that relying on clients to perform the deletion is error-prone, and Item 18 explains how the interface to the factory function can be modified to prevent common client errors, but such concerns are secondary here, because in this Item we address a more fundamental weakness of the code above: even if clients do everything right, there is no way to know how the program will behave.

The problem is that getTimeKeeper returns a pointer to a derived class object (e.g., AtomicClock), that object is being deleted via a base class pointer (i.e., a TimeKeeper* pointer), and the base class (TimeKeeper) has a *non-virtual destructor*. This is a recipe for disaster, because C++

specifies that when a derived class object is deleted through a pointer to a base class with a non-virtual destructor, results are undefined. What typically happens at runtime is that the derived part of the object is never destroyed. If getTimeKeeper were to return a pointer to an AtomicClock object, the AtomicClock part of the object (i.e., the data members declared in the AtomicClock class) would probably not be destroyed, nor would the AtomicClock destructor run. However, the base class part (i.e., the TimeKeeper part) typically would be destroyed, thus leading to a curious "partially destroyed" object. This is an excellent way to leak resources, corrupt data structures, and spend a lot of time with a debugger.

Eliminating the problem is simple: give the base class a virtual destructor. Then deleting a derived class object will do exactly what you want. It will destroy the entire object, including all its derived class parts:

```
class TimeKeeper {
public:
  TimeKeeper();
  virtual ~TimeKeeper();
  ...
};

TimeKeeper *ptk = getTimeKeeper();

...
delete ptk;                          // now behaves correctly
```

Base classes like TimeKeeper generally contain virtual functions other than the destructor, because the purpose of virtual functions is to allow customization of derived class implementations (see Item 34). For example, TimeKeeper might have a virtual function, getCurrentTime, which would be implemented differently in the various derived classes. Any class with virtual functions should almost certainly have a virtual destructor.

If a class does *not* contain virtual functions, that often indicates it is not meant to be used as a base class. When a class is not intended to be a base class, making the destructor virtual is usually a bad idea. Consider a class for representing points in two-dimensional space:

```
class Point {                        // a 2D point
public:
  Point(int xCoord, int yCoord);
  ~Point();

private:
  int x, y;
};
```

If an int occupies 32 bits, a Point object can typically fit into a 64-bit register. Furthermore, such a Point object can be passed as a 64-bit quantity to functions written in other languages, such as C or FORTRAN. If Point's destructor is made virtual, however, the situation changes.

The implementation of virtual functions requires that objects carry information that can be used at runtime to determine which virtual functions should be invoked on the object. This information typically takes the form of a pointer called a vptr ("virtual table pointer"). The vptr points to an array of function pointers called a vtbl ("virtual table"); each class with virtual functions has an associated vtbl. When a virtual function is invoked on an object, the actual function called is determined by following the object's vptr to a vtbl and then looking up the appropriate function pointer in the vtbl.

The details of how virtual functions are implemented are unimportant. What *is* important is that if the Point class contains a virtual function, objects of that type will increase in size. On a 32-bit architecture, they'll go from 64 bits (for the two ints) to 96 bits (for the ints plus the vptr); on a 64-bit architecture, they may go from 64 to 128 bits, because pointers on such architectures are 64 bits in size. Addition of a vptr to Point will thus increase its size by 50–100%! No longer can Point objects fit in a 64-bit register. Furthermore, Point objects in C++ can no longer look like the same structure declared in another language such as C, because their foreign language counterparts will lack the vptr. As a result, it is no longer possible to pass Points to and from functions written in other languages unless you explicitly compensate for the vptr, which is itself an implementation detail and hence unportable.

The bottom line is that gratuitously declaring all destructors virtual is just as wrong as never declaring them virtual. In fact, many people summarize the situation this way: declare a virtual destructor in a class if and only if that class contains at least one virtual function.

It is possible to get bitten by the non-virtual destructor problem even in the complete absence of virtual functions. For example, the standard string type contains no virtual functions, but misguided programmers sometimes use it as a base class anyway:

```
class SpecialString: public std::string {       // bad idea! std::string has a
  ...                                           // non-virtual destructor
};
```

At first glance, this may look innocuous, but if anywhere in an application you somehow convert a pointer-to-SpecialString into a pointer-to-

string and you then use delete on the string pointer, you are instantly transported to the realm of undefined behavior:

```
SpecialString *pss =new SpecialString("Impending Doom");
std::string *ps;
...
ps = pss;                          // SpecialString* ⇒ std::string*
...
delete ps;                         // undefined! In practice,
                                   // *ps's SpecialString resources
                                   // will be leaked, because the
                                   // SpecialString destructor won't
                                   // be called.
```

The same analysis applies to any class lacking a virtual destructor, including all the STL container types (e.g., vector, list, set, tr1::unordered_map (see Item 54), etc.). If you're ever tempted to inherit from a standard container or any other class with a non-virtual destructor, resist the temptation! (Unfortunately, C++ offers no derivation-prevention mechanism akin to Java's final classes or C#'s sealed classes.)

Occasionally it can be convenient to give a class a pure virtual destructor. Recall that pure virtual functions result in *abstract* classes — classes that can't be instantiated (i.e., you can't create objects of that type). Sometimes, however, you have a class that you'd like to be abstract, but you don't have any pure virtual functions. What to do? Well, because an abstract class is intended to be used as a base class, and because a base class should have a virtual destructor, and because a pure virtual function yields an abstract class, the solution is simple: declare a pure virtual destructor in the class you want to be abstract. Here's an example:

```
class AWOV {                       // AWOV = "Abstract w/o Virtuals"
public:
  virtual ~AWOV() = 0;             // declare pure virtual destructor
};
```

This class has a pure virtual function, so it's abstract, and it has a virtual destructor, so you won't have to worry about the destructor problem. There is one twist, however: you must provide a *definition* for the pure virtual destructor:

```
AWOV::~AWOV() {}                   // definition of pure virtual dtor
```

The way destructors work is that the most derived class's destructor is called first, then the destructor of each base class is called. Compil-

ers will generate a call to ~AWOV from its derived classes' destructors, so you have to be sure to provide a body for the function. If you don't, the linker will complain.

The rule for giving base classes virtual destructors applies only to *polymorphic* base classes — to base classes designed to allow the manipulation of derived class types through base class interfaces. TimeKeeper is a polymorphic base class, because we expect to be able to manipulate AtomicClock and WaterClock objects, even if we have only TimeKeeper pointers to them.

Not all base classes are designed to be used polymorphically. Neither the standard string type, for example, nor the STL container types are designed to be base classes at all, much less polymorphic ones. Some classes are designed to be used as base classes, yet are not designed to be used polymorphically. Such classes — examples include Uncopyable from Item 6 and input_iterator_tag from the standard library (see Item 47) — are not designed to allow the manipulation of derived class objects via base class interfaces. As a result, they don't need virtual destructors.

Things to Remember

- Polymorphic base classes should declare virtual destructors. If a class has any virtual functions, it should have a virtual destructor.
- Classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.

Item 8: Prevent exceptions from leaving destructors.

C++ doesn't prohibit destructors from emitting exceptions, but it certainly discourages the practice. With good reason. Consider:

```
class Widget {
public:
  ...
  ~Widget() { ... }                    // assume this might emit an exception
};

void doSomething()
{
  std::vector<Widget> v;
  ...
}                                      // v is automatically destroyed here
```

When the vector v is destroyed, it is responsible for destroying all the Widgets it contains. Suppose v has ten Widgets in it, and during destruction of the first one, an exception is thrown. The other nine

Widgets still have to be destroyed (otherwise any resources they hold would be leaked), so v should invoke their destructors. But suppose that during those calls, a second Widget destructor throws an exception. Now there are two simultaneously active exceptions, and that's one too many for C++. Depending on the precise conditions under which such pairs of simultaneously active exceptions arise, program execution either terminates or yields undefined behavior. In this example, it yields undefined behavior. It would yield equally undefined behavior using any other standard library container (e.g., list, set), any container in TR1 (see Item 54), or even an array. Not that containers or arrays are required to get into trouble. Premature program termination or undefined behavior can result from destructors emitting exceptions even without using containers and arrays. C++ does *not* like destructors that emit exceptions!

That's easy enough to understand, but what should you do if your destructor needs to perform an operation that may fail by throwing an exception? For example, suppose you're working with a class for database connections:

```
class DBConnection {
public:
  ...
  static DBConnection create();          // function to return
                                         // DBConnection objects; params
                                         // omitted for simplicity
  void close();                          // close connection; throw an
};                                       // exception if closing fails
```

To ensure that clients don't forget to call close on DBConnection objects, a reasonable idea would be to create a resource-managing class for DBConnection that calls close in its destructor. Such resource-managing classes are explored in detail in Chapter 3, but here, it's enough to consider what the destructor for such a class would look like:

```
class DBConn {                           // class to manage DBConnection
public:                                  // objects
  ...
  ~DBConn()                              // make sure database connections
  {                                      // are always closed
    db.close();
  }
private:
  DBConnection db;
};
```

That allows clients to program like this:

```
{                                       // open a block
  DBConn dbc(DBConnection::create());   // create DBConnection object
                                        // and turn it over to a DBConn
                                        // object to manage
  ...                                   // use the DBConnection object
                                        // via the DBConn interface
}                                       // at end of block, the DBConn
                                        // object is destroyed, thus
                                        // automatically calling close on
                                        // the DBConnection object
```

This is fine as long as the call to close succeeds, but if the call yields an exception, DBConn's destructor will propagate that exception, i.e., allow it to leave the destructor. That's a problem, because destructors that throw mean trouble.

There are two primary ways to avoid the trouble. DBConn's destructor could:

- **Terminate the program** if close throws, typically by calling abort:

```
DBConn::~DBConn()
{
  try { db.close(); }
  catch (...) {
    make log entry that the call to close failed;
    std::abort();
  }
}
```

  This is a reasonable option if the program cannot continue to run after an error is encountered during destruction. It has the advantage that if allowing the exception to propagate from the destructor would lead to undefined behavior, this prevents that from happening. That is, calling abort may forestall undefined behavior.

- **Swallow the exception** arising from the call to close:

```
DBConn::~DBConn()
{
  try { db.close(); }
  catch (...) {
    make log entry that the call to close failed;
  }
}
```

  In general, swallowing exceptions is a bad idea, because it suppresses important information — *something failed!* Sometimes, however, swallowing exceptions is preferable to running the risk of

premature program termination or undefined behavior. For this to be a viable option, the program must be able to reliably continue execution even after an error has been encountered and ignored.

Neither of these approaches is especially appealing. The problem with both is that the program has no way to react to the condition that led to close throwing an exception in the first place.

A better strategy is to design DBConn's interface so that its clients have an opportunity to react to problems that may arise. For example, DBConn could offer a close function itself, thus giving clients a chance to handle exceptions arising from that operation. It could also keep track of whether its DBConnection had been closed, closing it itself in the destructor if not. That would prevent a connection from leaking. If the call to close were to fail in the DBConnection destructor, however, we'd be back to terminating or swallowing:

```
class DBConn {
public:
  ...
  void close()                                  // new function for
  {                                             // client use
    db.close();
    closed = true;
  }

  ~DBConn()
  {
    if (!closed) {
      try {                                     // close the connection
        db.close();                             // if the client didn't
      }
      catch (...) {                             // if closing fails,
        make log entry that call to close failed; // note that and
        ...                                     // terminate or swallow
      }
    }
  }

private:
  DBConnection db;
  bool closed;
};
```

Moving the responsibility for calling close from DBConn's destructor to DBConn's client (with DBConn's destructor containing a "backup" call) may strike you as an unscrupulous shift of burden. You might even view it as a violation of Item 18's advice to make interfaces easy to use correctly. In fact, it's neither. If an operation may fail by throwing an exception and there may be a need to handle that exception, the exception *has to come from some non-destructor function*. That's

because destructors that emit exceptions are dangerous, always running the risk of premature program termination or undefined behavior. In this example, telling clients to call close themselves doesn't impose a burden on them; it gives them an opportunity to deal with errors they would otherwise have no chance to react to. If they don't find that opportunity useful (perhaps because they believe that no error will really occur), they can ignore it, relying on DBConn's destructor to call close for them. If an error occurs at that point — if close *does* throw — they're in no position to complain if DBConn swallows the exception or terminates the program. After all, they had first crack at dealing with the problem, and they chose not to use it.

Things to Remember

- Destructors should never emit exceptions. If functions called in a destructor may throw, the destructor should catch any exceptions, then swallow them or terminate the program.
- If class clients need to be able to react to exceptions thrown during an operation, the class should provide a regular (i.e., non-destructor) function that performs the operation.

Item 9: Never call virtual functions during construction or destruction.

I'll begin with the recap: you shouldn't call virtual functions during construction or destruction, because the calls won't do what you think, and if they did, you'd still be unhappy. If you're a recovering Java or C# programmer, pay close attention to this Item, because this is a place where those languages zig, while C++ zags.

Suppose you've got a class hierarchy for modeling stock transactions, e.g., buy orders, sell orders, etc. It's important that such transactions be auditable, so each time a transaction object is created, an appropriate entry needs to be created in an audit log. This seems like a reasonable way to approach the problem:

```
class Transaction {                          // base class for all
public:                                      // transactions
  Transaction();

  virtual void logTransaction() const = 0;   // make type-dependent
                                             // log entry
  ...
};
```

```
Transaction::Transaction()                          // implementation of
{                                                   // base class ctor
  ...
  logTransaction();                                 // as final action, log this
}                                                   // transaction

class BuyTransaction: public Transaction {          // derived class
public:
  virtual void logTransaction() const;              // how to log trans-
                                                    // actions of this type
  ...
};

class SellTransaction: public Transaction {         // derived class
public:
  virtual void logTransaction() const;              // how to log trans-
                                                    // actions of this type
  ...
};
```

Consider what happens when this code is executed:

```
BuyTransaction b;
```

Clearly a BuyTransaction constructor will be called, but first, a Transaction constructor must be called; base class parts of derived class objects are constructed before derived class parts are. The last line of the Transaction constructor calls the virtual function logTransaction, but this is where the surprise comes in. The version of logTransaction that's called is the one in Transaction, *not* the one in BuyTransaction — even though the type of object being created is BuyTransaction. During base class construction, virtual functions never go down into derived classes. Instead, the object behaves as if it were of the base type. Informally speaking, during base class construction, virtual functions aren't.

There's a good reason for this seemingly counterintuitive behavior. Because base class constructors execute before derived class constructors, derived class data members have not been initialized when base class constructors run. If virtual functions called during base class construction went down to derived classes, the derived class functions would almost certainly refer to local data members, but those data members would not yet have been initialized. That would be a non-stop ticket to undefined behavior and late-night debugging sessions. Calling down to parts of an object that have not yet been initialized is inherently dangerous, so C++ gives you no way to do it.

It's actually more fundamental than that. During base class construction of a derived class object, the type of the object *is* that of the base

class. Not only do virtual functions resolve to the base class, but the parts of the language using runtime type information (e.g., dynamic_cast (see Item 27) and typeid) treat the object as a base class type. In our example, while the Transaction constructor is running to initialize the base class part of a BuyTransaction object, the object is of type Transaction. That's how every part of C++ will treat it, and the treatment makes sense: the BuyTransaction-specific parts of the object haven't been initialized yet, so it's safest to treat them as if they didn't exist. An object doesn't become a derived class object until execution of a derived class constructor begins.

The same reasoning applies during destruction. Once a derived class destructor has run, the object's derived class data members assume undefined values, so C++ treats them as if they no longer exist. Upon entry to the base class destructor, the object becomes a base class object, and all parts of C++ — virtual functions, dynamic_casts, etc., — treat it that way.

In the example code above, the Transaction constructor made a direct call to a virtual function, a clear and easy-to-see violation of this Item's guidance. The violation is so easy to see, some compilers issue a warning about it. (Others don't. See Item 53 for a discussion of warnings.) Even without such a warning, the problem would almost certainly become apparent before runtime, because the logTransaction function is pure virtual in Transaction. Unless it had been defined (unlikely, but possible — see Item 34), the program wouldn't link: the linker would be unable to find the necessary implementation of Transaction::logTransaction.

It's not always so easy to detect calls to virtual functions during construction or destruction. If Transaction had multiple constructors, each of which had to perform some of the same work, it would be good software engineering to avoid code replication by putting the common initialization code, including the call to logTransaction, into a private non-virtual initialization function, say, init:

```
class Transaction {
public:
  Transaction()
  { init(); }                                  // call to non-virtual...
  virtual void logTransaction() const = 0;
  ...
private:
  void init()
  {
    ...
    logTransaction();                          // ...that calls a virtual!
  }
};
```

This code is conceptually the same as the earlier version, but it's more insidious, because it will typically compile and link without complaint. In this case, because logTransaction is pure virtual in Transaction, most runtime systems will abort the program when the pure virtual is called (typically issuing a message to that effect). However, if logTransaction were a "normal" virtual function (i.e., not pure virtual) with an implementation in Transaction, that version would be called, and the program would merrily trot along, leaving you to figure out why the wrong version of logTransaction was called when a derived class object was created. The only way to avoid this problem is to make sure that none of your constructors or destructors call virtual functions on the object being created or destroyed and that all the functions they call obey the same constraint.

But how *do* you ensure that the proper version of logTransaction is called each time an object in the Transaction hierarchy is created? Clearly, calling a virtual function on the object from the Transaction constructor(s) is the wrong way to do it.

There are different ways to approach this problem. One is to turn logTransaction into a non-virtual function in Transaction, then require that derived class constructors pass the necessary log information to the Transaction constructor. That function can then safely call the non-virtual logTransaction. Like this:

```
class Transaction {
public:
  explicit Transaction(const std::string& logInfo);
  void logTransaction(const std::string& logInfo) const;    // now a non-
                                                            // virtual func
  ...
};

Transaction::Transaction(const std::string& logInfo)
{
  ...
  logTransaction(logInfo);                                  // now a non-
}                                                           // virtual call

class BuyTransaction: public Transaction {
public:
  BuyTransaction( parameters )
  : Transaction(createLogString( parameters ))              // pass log info
  { ... }                                                   // to base class
  ...                                                       // constructor

private:
  static std::string createLogString( parameters );
};
```

In other words, since you can't use virtual functions to call down from base classes during construction, you can compensate by having derived classes pass necessary construction information up to base class constructors instead.

In this example, note the use of the (private) static function createLogString in BuyTransaction. Using a helper function to create a value to pass to a base class constructor is often more convenient (and more readable) than going through contortions in the member initialization list to give the base class what it needs. By making the function static, there's no danger of accidentally referring to the nascent BuyTransaction object's as-yet-uninitialized data members. That's important, because the fact that those data members will be in an undefined state is why calling virtual functions during base class construction and destruction doesn't go down into derived classes in the first place.

Things to Remember

- Don't call virtual functions during construction or destruction, because such calls will never go to a more derived class than that of the currently executing constructor or destructor.

Item 10: Have assignment operators return a reference to *this.

One of the interesting things about assignments is that you can chain them together:

```
int x, y, z;
x = y = z = 15;                    // chain of assignments
```

Also interesting is that assignment is right-associative, so the above assignment chain is parsed like this:

```
x = (y = (z = 15));
```

Here, 15 is assigned to z, then the result of that assignment (the updated z) is assigned to y, then the result of that assignment (the updated y) is assigned to x.

The way this is implemented is that assignment returns a reference to its left-hand argument, and that's the convention you should follow when you implement assignment operators for your classes:

```
class Widget {
public:
  ...
```

```
  Widget& operator=(const Widget& rhs)     // return type is a reference to
  {                                        // the current class
    ...
    return *this;                          // return the left-hand object
  }
  ...
};
```

This convention applies to all assignment operators, not just the standard form shown above. Hence:

```
class Widget {
public:
  ...
  Widget& operator+=(const Widget& rhs)    // the convention applies to
  {                                        // +=, -=, *=, etc.
    ...
    return *this;
  }

  Widget& operator=(int rhs)               // it applies even if the
  {                                        // operator's parameter type
    ...                                    // is unconventional
    return *this;
  }
  ...
};
```

This is only a convention; code that doesn't follow it will compile. However, the convention is followed by all the built-in types as well as by all the types in (or soon to be in — see Item 54) the standard library (e.g., string, vector, complex, tr1::shared_ptr, etc.). Unless you have a good reason for doing things differently, don't.

Things to Remember

✦ Have assignment operators return a reference to *this.

Item 11: Handle assignment to self in operator=.

An assignment to self occurs when an object is assigned to itself:

```
class Widget { ... };
Widget w;
...
w = w;                                     // assignment to self
```

This looks silly, but it's legal, so rest assured that clients will do it. Besides, assignment isn't always so recognizable. For example,

```
a[i] = a[j];                    // potential assignment to self
```

is an assignment to self if i and j have the same value, and

```
*px = *py;                      // potential assignment to self
```

is an assignment to self if px and py happen to point to the same thing. These less obvious assignments to self are the result of *aliasing*: having more than one way to refer to an object. In general, code that operates on references or pointers to multiple objects of the same type needs to consider that the objects might be the same. In fact, the two objects need not even be declared to be of the same type if they're from the same hierarchy, because a base class reference or pointer can refer or point to an object of a derived class type:

```
class Base { ... };
class Derived: public Base { ... };
void doSomething(const Base& rb,    // rb and *pd might actually be
                 Derived* pd);      // the same object
```

If you follow the advice of Items 13 and 14, you'll always use objects to manage resources, and you'll make sure that the resource-managing objects behave well when copied. When that's the case, your assignment operators will probably be self-assignment-safe without your having to think about it. If you try to manage resources yourself, however (which you'd certainly have to do if you were writing a resource-managing class), you can fall into the trap of accidentally releasing a resource before you're done using it. For example, suppose you create a class that holds a raw pointer to a dynamically allocated bitmap:

```
class Bitmap { ... };
class Widget {
  ...
private:
  Bitmap *pb;                   // ptr to a heap-allocated object
};
```

Here's an implementation of operator= that looks reasonable on the surface but is unsafe in the presence of assignment to self. (It's also not exception-safe, but we'll deal with that in a moment.)

```
Widget&
Widget::operator=(const Widget& rhs)  // unsafe impl. of operator=
{
  delete pb;                    // stop using current bitmap
  pb = new Bitmap(*rhs.pb);     // start using a copy of rhs's bitmap

  return *this;                 // see Item 10
}
```

The self-assignment problem here is that inside operator=, *this (the target of the assignment) and rhs could be the same object. When they are, the delete not only destroys the bitmap for the current object, it destroys the bitmap for rhs, too. At the end of the function, the Widget — which should not have been changed by the assignment to self — finds itself holding a pointer to a deleted object![†]

The traditional way to prevent this error is to check for assignment to self via an *identity test* at the top of operator=:

```
Widget& Widget::operator=(const Widget& rhs)
{
  if (this == &rhs) return *this;          // identity test: if a self-assignment,
                                           // do nothing
  delete pb;
  pb = new Bitmap(*rhs.pb);

  return *this;
}
```

This works, but I mentioned above that the previous version of operator= wasn't just self-assignment-unsafe, it was also exception-unsafe, and this version continues to have exception trouble. In particular, if the "new Bitmap" expression yields an exception (either because there is insufficient memory for the allocation or because Bitmap's copy constructor throws one), the Widget will end up holding a pointer to a deleted Bitmap. Such pointers are toxic. You can't safely delete them. You can't even safely read them. About the only safe thing you can do with them is spend lots of debugging energy figuring out where they came from.

Happily, making operator= exception-safe typically renders it self-assignment-safe, too. As a result, it's increasingly common to deal with issues of self-assignment by ignoring them, focusing instead on achieving exception safety. Item 29 explores exception safety in depth, but in this Item, it suffices to observe that in many cases, a careful ordering of statements can yield exception-safe (and self-assignment-safe) code. Here, for example, we just have to be careful not to delete pb until after we've copied what it points to:

```
Widget& Widget::operator=(const Widget& rhs)
{
  Bitmap *pOrig = pb;                      // remember original pb
  pb = new Bitmap(*rhs.pb);                // make pb point to a copy of *pb
  delete pOrig;                            // delete the original pb

  return *this;
}
```

† Probably. C++ implementations are permitted to change the value of a deleted pointer (e.g., to null or some other special bit pattern), but I am unaware of any that do.

Now, if "new Bitmap" throws an exception, pb (and the Widget it's inside of) remains unchanged. Even without the identity test, this code handles assignment to self, because we make a copy of the original bitmap, delete the original bitmap, then point to the copy we made. It may not be the most efficient way to handle self-assignment, but it does work.

If you're concerned about efficiency, you could put the identity test back at the top of the function. Before doing that, however, ask yourself how often you expect self-assignments to occur, because the test isn't free. It makes the code (both source and object) a bit bigger, and it introduces a branch into the flow of control, both of which can decrease runtime speed. The effectiveness of instruction prefetching, caching, and pipelining can be reduced, for example.

An alternative to manually ordering statements in operator= to make sure the implementation is both exception- and self-assignment-safe is to use the technique known as "copy and swap." This technique is closely associated with exception safety, so it's described in Item 29. However, it's a common enough way to write operator= that it's worth seeing what such an implementation often looks like:

```
class Widget {
  ...
  void swap(Widget& rhs);          // exchange *this's and rhs's data;
  ...                              // see Item 29 for details
};

Widget& Widget::operator=(const Widget& rhs)
{
  Widget temp(rhs);                // make a copy of rhs's data
  swap(temp);                      // swap *this's data with the copy's
  return *this;
}
```

A variation on this theme takes advantage of the facts that (1) a class's copy assignment operator may be declared to take its argument by value and (2) passing something by value makes a *copy* of it (see Item 20):

```
Widget& Widget::operator=(Widget rhs)   // rhs is a copy of the object
{                                       // passed in — note pass by val
  swap(rhs);                            // swap *this's data with
                                        // the copy's
  return *this;
}
```

Personally, I worry that this approach sacrifices clarity at the altar of cleverness, but by moving the copying operation from the body of the function to construction of the parameter, it's a fact that compilers can sometimes generate more efficient code.

Things to Remember

- Make sure operator= is well-behaved when an object is assigned to itself. Techniques include comparing addresses of source and target objects, careful statement ordering, and copy-and-swap.
- Make sure that any function operating on more than one object behaves correctly if two or more of the objects are the same.

Item 12: Copy all parts of an object.

In well-designed object-oriented systems that encapsulate the internal parts of objects, only two functions copy objects: the aptly named copy constructor and copy assignment operator. We'll call these the *copying functions*. Item 5 observes that compilers will generate the copying functions, if needed, and it explains that the compiler-generated versions do precisely what you'd expect: they copy all the data of the object being copied.

When you declare your own copying functions, you are indicating to compilers that there is something about the default implementations you don't like. Compilers seem to take offense at this, and they retaliate in a curious fashion: they don't tell you when your implementations are almost certainly wrong.

Consider a class representing customers, where the copying functions have been manually written so that calls to them are logged:

```
void logCall(const std::string& funcName);          // make a log entry

class Customer {
public:
  ...
  Customer(const Customer& rhs);
  Customer& operator=(const Customer& rhs);
  ...

private:
  std::string name;
};
```

```
Customer::Customer(const Customer& rhs)
: name(rhs.name)                                    // copy rhs's data
{
  logCall("Customer copy constructor");
}

Customer& Customer::operator=(const Customer& rhs)
{
  logCall("Customer copy assignment operator");
  name = rhs.name;                                  // copy rhs's data
  return *this;                                     // see Item 10
}
```

Everything here looks fine, and in fact everything is fine — until another data member is added to Customer:

```
class Date { ... };                                 // for dates in time

class Customer {
public:
  ...                                               // as before

private:
  std::string name;
  Date lastTransaction;
};
```

At this point, the existing copying functions are performing a *partial copy*: they're copying the customer's name, but not its lastTransaction. Yet most compilers say nothing about this, not even at maximal warning level (see also Item 53). That's their revenge for your writing the copying functions yourself. You reject the copying functions they'd write, so they don't tell you if your code is incomplete. The conclusion is obvious: if you add a data member to a class, you need to make sure that you update the copying functions, too. (You'll also need to update all the constructors (see Items 4 and 45) as well as any non-standard forms of operator= in the class (Item 10 gives an example). If you forget, compilers are unlikely to remind you.)

One of the most insidious ways this issue can arise is through inheritance. Consider:

```
class PriorityCustomer: public Customer {           // a derived class
public:
  ...
  PriorityCustomer(const PriorityCustomer& rhs);
  PriorityCustomer& operator=(const PriorityCustomer& rhs);
  ...

private:
  int priority;
};
```

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
: priority(rhs.priority)
{
  logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
  logCall("PriorityCustomer copy assignment operator");

  priority = rhs.priority;

  return *this;
}
```

PriorityCustomer's copying functions look like they're copying everything in PriorityCustomer, but look again. Yes, they copy the data member that PriorityCustomer declares, but every PriorityCustomer also contains a copy of the data members it inherits from Customer, and those data members are not being copied at all! PriorityCustomer's copy constructor specifies no arguments to be passed to its base class constructor (i.e., it makes no mention of Customer on its member initialization list), so the Customer part of the PriorityCustomer object will be initialized by the Customer constructor taking no arguments — by the default constructor. (Assuming it has one. If not, the code won't compile.) That constructor will perform a *default* initialization for name and lastTransaction.

The situation is only slightly different for PriorityCustomer's copy assignment operator. It makes no attempt to modify its base class data members in any way, so they'll remain unchanged.

Any time you take it upon yourself to write copying functions for a derived class, you must take care to also copy the base class parts. Those parts are typically private, of course (see Item 22), so you can't access them directly. Instead, derived class copying functions must invoke their corresponding base class functions:

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
: Customer(rhs),                          // invoke base class copy ctor
  priority(rhs.priority)
{
  logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
  logCall("PriorityCustomer copy assignment operator");

  Customer::operator=(rhs);               // assign base class parts
  priority = rhs.priority;

  return *this;
}
```

The meaning of "copy all parts" in this Item's title should now be clear. When you're writing a copying function, be sure to (1) copy all local data members and (2) invoke the appropriate copying function in all base classes, too.

In practice, the two copying functions will often have similar bodies, and this may tempt you to try to avoid code duplication by having one function call the other. Your desire to avoid code duplication is laudable, but having one copying function call the other is the wrong way to achieve it.

It makes no sense to have the copy assignment operator call the copy constructor, because you'd be trying to construct an object that already exists. This is so nonsensical, there's not even a syntax for it. There are syntaxes that *look* like you're doing it, but you're not; and there are syntaxes that *do* do it in a backwards kind of way, but they corrupt your object under some conditions. So I'm not going to show you any of those syntaxes. Simply accept that having the copy assignment operator call the copy constructor is something you don't want to do.

Trying things the other way around — having the copy constructor call the copy assignment operator — is equally nonsensical. A constructor initializes new objects, but an assignment operator applies only to objects that have already been initialized. Performing an assignment on an object under construction would mean doing something to a not-yet-initialized object that makes sense only for an initialized object. Nonsense! Don't try it.

Instead, if you find that your copy constructor and copy assignment operator have similar code bodies, eliminate the duplication by creating a third member function that both call. Such a function is typically private and is often named `init`. This strategy is a safe, proven way to eliminate code duplication in copy constructors and copy assignment operators.

Things to Remember

- Copying functions should be sure to copy all of an object's data members and all of its base class parts.
- Don't try to implement one of the copying functions in terms of the other. Instead, put common functionality in a third function that both call.

3

Resource Management

A resource is something that, once you're done using it, you need to return to the system. If you don't, bad things happen. In C++ programs, the most commonly used resource is dynamically allocated memory (if you allocate memory and never deallocate it, you've got a memory leak), but memory is only one of many resources you must manage. Other common resources include file descriptors, mutex locks, fonts and brushes in graphical user interfaces (GUIs), database connections, and network sockets. Regardless of the resource, it's important that it be released when you're finished with it.

Trying to ensure this by hand is difficult under any conditions, but when you consider exceptions, functions with multiple return paths, and maintenance programmers modifying software without fully comprehending the impact of their changes, it becomes clear that ad hoc ways of dealing with resource management aren't sufficient.

This chapter begins with a straightforward object-based approach to resource management built on C++'s support for constructors, destructors, and copying operations. Experience has shown that disciplined adherence to this approach can all but eliminate resource management problems. The chapter then moves on to Items dedicated specifically to memory management. These latter Items complement the more general Items that come earlier, because objects that manage memory have to know how to do it properly.

Item 13: Use objects to manage resources.

Suppose we're working with a library for modeling investments (e.g., stocks, bonds, etc.), where the various investment types inherit from a root class Investment:

```
class Investment { ... };              // root class of hierarchy of
                                       // investment types
```

Further suppose that the way the library provides us with specific Investment objects is through a factory function (see Item 7):

```
Investment* createInvestment();   // return ptr to dynamically allocated
                                  // object in the Investment hierarchy;
                                  // the caller must delete it
                                  // (parameters omitted for simplicity)
```

As the comment indicates, callers of createInvestment are responsible for deleting the object that function returns when they are done with it. Consider, then, a function f written to fulfill this obligation:

```
void f()
{
  Investment *pInv = createInvestment();      // call factory function
  ...                                         // use pInv
  delete pInv;                                // release object
}
```

This looks okay, but there are several ways f could fail to delete the investment object it gets from createInvestment. There might be a premature return statement somewhere inside the "..." part of the function. If such a return were executed, control would never reach the delete statement. A similar situation would arise if the uses of createInvestment and delete were in a loop, and the loop was prematurely exited by a break or goto statement. Finally, some statement inside the "..." might throw an exception. If so, control would again not get to the delete. Regardless of how the delete were to be skipped, we'd leak not only the memory containing the investment object but also any resources held by that object.

Of course, careful programming could prevent these kinds of errors, but think about how the code might change over time. As the software gets maintained, somebody might add a return or continue statement without fully grasping the repercussions on the rest of the function's resource management strategy. Even worse, the "..." part of f might call a function that never used to throw an exception but suddenly starts doing so after it has been "improved." Relying on f always getting to its delete statement simply isn't viable.

To make sure that the resource returned by createInvestment is always released, we need to put that resource inside an object whose destructor will automatically release the resource when control leaves f. In fact, that's half the idea behind this Item: by putting resources inside objects, we can rely on C++'s automatic destructor invocation to make sure that the resources are released. (We'll discuss the other half of the idea in a moment.)

Many resources are dynamically allocated on the heap, are used only within a single block or function, and should be released when control leaves that block or function. The standard library's auto_ptr is tailor-made for this kind of situation. auto_ptr is a pointer-like object (a *smart pointer*) whose destructor automatically calls delete on what it points to. Here's how to use auto_ptr to prevent f's potential resource leak:

```
void f()
{
  std::auto_ptr<Investment> pInv(createInvestment());   // call factory
                                                        // function
  ...                                                   // use pInv as
                                                        // before
}                                                       // automatically
                                                        // delete pInv via
                                                        // auto_ptr's dtor
```

This simple example demonstrates the two critical aspects of using objects to manage resources:

- **Resources are acquired and immediately turned over to resource-managing objects.** Above, the resource returned by createInvestment is used to initialize the auto_ptr that will manage it. In fact, the idea of using objects to manage resources is often called *Resource Acquisition Is Initialization* (RAII), because it's so common to acquire a resource and initialize a resource-managing object in the same statement. Sometimes acquired resources are *assigned* to resource-managing objects instead of initializing them, but either way, every resource is immediately turned over to a resource-managing object at the time the resource is acquired.
- **Resource-managing objects use their destructors to ensure that resources are released.** Because destructors are called automatically when an object is destroyed (e.g., when an object goes out of scope), resources are correctly released, regardless of how control leaves a block. Things can get tricky when the act of releasing resources can lead to exceptions being thrown, but that's a matter addressed by Item 8, so we'll not worry about it here.

Because an auto_ptr automatically deletes what it points to when the auto_ptr is destroyed, it's important that there never be more than one auto_ptr pointing to an object. If there were, the object would be deleted more than once, and that would put your program on the fast track to undefined behavior. To prevent such problems, auto_ptrs have an unusual characteristic: copying them (via copy constructor or copy

assignment operator) sets them to null, and the copying pointer assumes sole ownership of the resource!

```
std::auto_ptr<Investment>                    // plnv1 points to the
  plnv1(createInvestment());                 // object returned from
                                             // createInvestment

std::auto_ptr<Investment> plnv2(plnv1);      // plnv2 now points to the
                                             // object; plnv1 is now null

plnv1 = plnv2;                               // now plnv1 points to the
                                             // object, and plnv2 is null
```

This odd copying behavior, plus the underlying requirement that resources managed by auto_ptrs must never have more than one auto_ptr pointing to them, means that auto_ptrs aren't the best way to manage all dynamically allocated resources. For example, STL containers require that their contents exhibit "normal" copying behavior, so containers of auto_ptr aren't allowed.

An alternative to auto_ptr is a *reference-counting smart pointer* (RCSP). An RCSP is a smart pointer that keeps track of how many objects point to a particular resource and automatically deletes the resource when nobody is pointing to it any longer. As such, RCSPs offer behavior that is similar to that of garbage collection. Unlike garbage collection, however, RCSPs can't break cycles of references (e.g., two otherwise unused objects that point to one another).

TR1's tr1::shared_ptr (see Item 54) is an RCSP, so you could write f this way:

```
void f()
{
  ...
  std::tr1::shared_ptr<Investment>
    plnv(createInvestment());                // call factory function
  ...                                        // use plnv as before
}                                            // automatically delete
                                             // plnv via shared_ptr's dtor
```

This code looks almost the same as that employing auto_ptr, but copying shared_ptrs behaves much more naturally:

```
void f()
{
  ...
  std::tr1::shared_ptr<Investment>           // plnv1 points to the
    plnv1(createInvestment());               // object returned from
                                             // createInvestment
```