



Effective C++ Third Edition

55 Specific Ways to Improve
Your Programs and Designs

Scott Meyers



FREE SAMPLE CHAPTER

SHARE WITH OTHERS



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, Bleeding 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++*

Effective C++

Third Edition

Addison-Wesley Professional Computing Series

Brian W. Kernighan, Consulting Editor

Matthew H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*

David R. Butenhof, *Programming with POSIX® Threads*

Brent Callaghan, *NFS Illustrated*

Tom Cargill, *C++ Programming Style*

William R. Cheswick/Steven M. Bellovin/Aviel D. Rubin, *Firewalls and Internet Security, Second Edition: Repelling the Wily Hacker*

David A. Curry, *UNIX® System Security: A Guide for Users and System Administrators*

Stephen C. Dewhurst, *C++ Gotchas: Avoiding Common Problems in Coding and Design*

Dan Farmer/Wietse Venema, *Forensic Discovery*

Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*

Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides, *Design Patterns CD: Elements of Reusable Object-Oriented Software*

Peter Hagggar, *Practical Java™ Programming Language Guide*

David R. Hanson, *C Interfaces and Implementations: Techniques for Creating Reusable Software*

Mark Harrison/Michael McLennan, *Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk*

Michi Henning/Steve Vinoski, *Advanced CORBA® Programming with C++*

Brian W. Kernighan/Rob Pike, *The Practice of Programming*

S. Keshav, *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*

John Lakos, *Large-Scale C++ Software Design*

Scott Meyers, *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs*

Scott Meyers, *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*

Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*

Scott Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*

Robert B. Murray, *C++ Strategies and Tactics*

David R. Musser/Gillmer J. Derge/Atul Saini, *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*

John K. Ousterhout, *Tcl and the Tk Toolkit*

Craig Partridge, *Gigabit Networking*

Radia Perlman, *Interconnections, Second Edition: Bridges, Routers, Switches, and Internetworking Protocols*

Stephen A. Rago, *UNIX® System V Network Programming*

Eric S. Raymond, *The Art of UNIX Programming*

Marc J. Rochkind, *Advanced UNIX Programming, Second Edition*

Curt Schimmel, *UNIX® Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*

W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*

W. Richard Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX® Domain Protocols*

W. Richard Stevens/Bill Fenner/Andrew M. Rudoff, *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*

W. Richard Stevens/Stephen A. Rago, *Advanced Programming in the UNIX® Environment, Second Edition*

W. Richard Stevens/Gary R. Wright, *TCP/IP Illustrated Volumes 1-3 Boxed Set*

John Viega/Gary McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*

Gary R. Wright/W. Richard Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*

Ruixi Yuan/W. Timothy Strayer, *Virtual Private Networks: Technologies and Solutions*

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

This e-book reproduces in electronic form the printed book content of *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers. Copyright © 2005 by Pearson Education, Inc. ISBN: 0-321-33487-6.

LICENSE FOR PERSONAL USE: For the convenience of readers, this e-book is licensed and sold in its PDF version without any digital rights management (DRM) applied. Purchasers of the PDF version may, for their personal use only, install additional copies on multiple devices and copy or print excerpts for themselves. The duplication, distribution, transfer, or sharing of this e-book's content for any purpose other than the purchaser's personal use, in whole or in part, by any means, is strictly prohibited.

PERSONALIZATION NOTICE: To discourage unauthorized uses of this e-book and thereby allow its publication without DRM, each copy of the PDF version identifies its purchaser. To encourage a DRM-free policy, please protect your files from access by others.

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

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

DISCOUNTS AND SITE LICENSES: The publisher offers discounted prices on this e-book when purchased with its corresponding printed book or with other e-books by Scott Meyers. The publisher also offers site licenses for these e-books (not available in some countries). For more information, please visit: www.ScottMeyers-EBooks.com or www.informit.com/aw.

Copyright © 2008 by Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

E-book ISBN 13: 978-0-321-51582-7

E-book ISBN 10: 0-321-51582-X

Second e-book release, April 2011 (essentially identical to the 11th Paper Printing).

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

Contents

Preface	xv
Acknowledgments	xvii
Introduction	1
Chapter 1: Accustoming Yourself to C++	11
Item 1: View C++ as a federation of languages.	11
Item 2: Prefer consts, enums, and inlines to #defines.	13
Item 3: Use const whenever possible.	17
Item 4: Make sure that objects are initialized before they're used.	26
Chapter 2: Constructors, Destructors, and Assignment Operators	34
Item 5: Know what functions C++ silently writes and calls.	34
Item 6: Explicitly disallow the use of compiler-generated functions you do not want.	37
Item 7: Declare destructors virtual in polymorphic base classes.	40
Item 8: Prevent exceptions from leaving destructors.	44
Item 9: Never call virtual functions during construction or destruction.	48
Item 10: Have assignment operators return a reference to *this.	52
Item 11: Handle assignment to self in operator=.	53
Item 12: Copy all parts of an object.	57
Chapter 3: Resource Management	61
Item 13: Use objects to manage resources.	61

Item 14:	Think carefully about copying behavior in resource-managing classes.	66
Item 15:	Provide access to raw resources in resource-managing classes.	69
Item 16:	Use the same form in corresponding uses of new and delete.	73
Item 17:	Store newed objects in smart pointers in standalone statements.	75
Chapter 4: Designs and Declarations		78
Item 18:	Make interfaces easy to use correctly and hard to use incorrectly.	78
Item 19:	Treat class design as type design.	84
Item 20:	Prefer pass-by-reference-to-const to pass-by-value.	86
Item 21:	Don't try to return a reference when you must return an object.	90
Item 22:	Declare data members private.	94
Item 23:	Prefer non-member non-friend functions to member functions.	98
Item 24:	Declare non-member functions when type conversions should apply to all parameters.	102
Item 25:	Consider support for a non-throwing swap.	106
Chapter 5: Implementations		113
Item 26:	Postpone variable definitions as long as possible.	113
Item 27:	Minimize casting.	116
Item 28:	Avoid returning "handles" to object internals.	123
Item 29:	Strive for exception-safe code.	127
Item 30:	Understand the ins and outs of inlining.	134
Item 31:	Minimize compilation dependencies between files.	140
Chapter 6: Inheritance and Object-Oriented Design		149
Item 32:	Make sure public inheritance models "is-a."	150
Item 33:	Avoid hiding inherited names.	156
Item 34:	Differentiate between inheritance of interface and inheritance of implementation.	161
Item 35:	Consider alternatives to virtual functions.	169
Item 36:	Never redefine an inherited non-virtual function.	178

<i>Effective C++</i>	Contents	xiii
Item 37: Never redefine a function's inherited default parameter value.		180
Item 38: Model "has-a" or "is-implemented-in-terms-of" through composition.		184
Item 39: Use private inheritance judiciously.		187
Item 40: Use multiple inheritance judiciously.		192
Chapter 7: Templates and Generic Programming		199
Item 41: Understand implicit interfaces and compile-time polymorphism.		199
Item 42: Understand the two meanings of typename.		203
Item 43: Know how to access names in templated base classes.		207
Item 44: Factor parameter-independent code out of templates.		212
Item 45: Use member function templates to accept "all compatible types."		218
Item 46: Define non-member functions inside templates when type conversions are desired.		222
Item 47: Use traits classes for information about types.		226
Item 48: Be aware of template metaprogramming.		233
Chapter 8: Customizing new and delete		239
Item 49: Understand the behavior of the new-handler.		240
Item 50: Understand when it makes sense to replace new and delete.		247
Item 51: Adhere to convention when writing new and delete.		252
Item 52: Write placement delete if you write placement new.		256
Chapter 9: Miscellany		262
Item 53: Pay attention to compiler warnings.		262
Item 54: Familiarize yourself with the standard library, including TR1.		263
Item 55: Familiarize yourself with Boost.		269
Appendix A: Beyond <i>Effective C++</i>		273
Appendix B: Item Mappings Between Second and Third Editions		277
Index		280

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/>.

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#Best-Practices 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 Brauneegg, 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 Creutz, 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 Viciano, 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. Fehér.

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 Wieggers 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 Fehér, 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, Jiongxiang 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, Brian Johnson, Joe Suzow, Effeer Chen, Nate Kohl, Zachary Cohen, Owen Chu, and Molly Sharp.

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
  while ((number /= 10) != 0) ++digitsSoFar; // in its parameter.)
  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.


```

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
    ...
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:  
    ...
```

Here I've switched from an object of type `string` to an object of type `Widget` to avoid any preconceptions about the cost of performing a construction, destruction, or assignment for the object.

In terms of `Widget` operations, the costs of these two approaches are as follows:

- Approach A: 1 constructor + 1 destructor + n assignments.
- Approach B: n constructors + n destructors.

For classes where an assignment costs less than a constructor-destructor pair, Approach A is generally more efficient. This is especially the case as n gets large. Otherwise, Approach B is probably better. Furthermore, Approach A makes the name `w` visible in a larger scope (the one containing the loop) than Approach B, something that's contrary to program comprehensibility and maintainability. As a result, unless you know that (1) assignment is less expensive than a constructor-destructor pair and (2) you're dealing with a performance-sensitive part of your code, you should default to using Approach B.

Things to Remember

- ♦ Postpone variable definitions as long as possible. It increases program clarity and improves program efficiency.

Item 27: Minimize casting.

The rules of C++ are designed to guarantee that type errors are impossible. In theory, if your program compiles cleanly, it's not trying to perform any unsafe or nonsensical operations on any objects. This is a valuable guarantee. You don't want to forgo it lightly.

Unfortunately, casts subvert the type system. That can lead to all kinds of trouble, some easy to recognize, some extraordinarily subtle. If you're coming to C++ from C, Java, or C#, take note, because casting in those languages is more necessary and less dangerous than in C++. But C++ is not C. It's not Java. It's not C#. In this language, casting is a feature you want to approach with great respect.

Let's begin with a review of casting syntax, because there are usually three different ways to write the same cast. C-style casts look like this:

```
(T) expression // cast expression to be of type T
```

Function-style casts use this syntax:

```
T(expression) // cast expression to be of type T
```

There is no difference in meaning between these forms; it's purely a matter of where you put the parentheses. I call these two forms *old-style casts*.

C++ also offers four new cast forms (often called *new-style* or *C++-style casts*):

```
const_cast<T>(expression)
dynamic_cast<T>(expression)
reinterpret_cast<T>(expression)
static_cast<T>(expression)
```

Each serves a distinct purpose:

- `const_cast` is typically used to cast away the constness of objects. It is the only C++-style cast that can do this.
- `dynamic_cast` is primarily used to perform “safe downcasting,” i.e., to determine whether an object is of a particular type in an inheritance hierarchy. It is the only cast that cannot be performed using the old-style syntax. It is also the only cast that may have a significant runtime cost. (I'll provide details on this a bit later.)
- `reinterpret_cast` is intended for low-level casts that yield implementation-dependent (i.e., unportable) results, e.g., casting a pointer to an int. Such casts should be rare outside low-level code. I use it only once in this book, and that's only when discussing how you might write a debugging allocator for raw memory (see [Item 50](#)).
- `static_cast` can be used to force implicit conversions (e.g., non-const object to const object (as in [Item 3](#)), int to double, etc.). It can also be used to perform the reverse of many such conversions (e.g., void* pointers to typed pointers, pointer-to-base to pointer-to-derived), though it cannot cast from const to non-const objects. (Only `const_cast` can do that.)

The old-style casts continue to be legal, but the new forms are preferable. First, they're much easier to identify in code (both for humans and for tools like `grep`), thus simplifying the process of finding places in the code where the type system is being subverted. Second, the more narrowly specified purpose of each cast makes it possible for compilers to diagnose usage errors. For example, if you try to cast away constness using a new-style cast other than `const_cast`, your code won't compile.

About the only time I use an old-style cast is when I want to call an explicit constructor to pass an object to a function. For example:

```

class Widget {
public:
    explicit Widget(int size);
    ...
};

void doSomeWork(const Widget& w);
doSomeWork(Widget(15));           // create Widget from int
                                   // with function-style cast

doSomeWork(static_cast<Widget>(15)); // create Widget from int
                                   // with C++-style cast

```

Somehow, deliberate object creation doesn't "feel" like a cast, so I'd probably use the function-style cast instead of the `static_cast` in this case. (They do exactly the same thing here: create a temporary `Widget` object to pass to `doSomeWork`.) Then again, code that leads to a core dump usually feels pretty reasonable when you write it, so perhaps you'd best ignore feelings and use new-style casts all the time.

Many programmers believe that casts do nothing but tell compilers to treat one type as another, but this is mistaken. Type conversions of any kind (either explicit via casts or implicit by compilers) often lead to code that is executed at runtime. For example, in this code fragment,

```

int x, y;
...
double d = static_cast<double>(x)/y; // divide x by y, but use
                                       // floating point division

```

the cast of the `int x` to a `double` almost certainly generates code, because on most architectures, the underlying representation for an `int` is different from that for a `double`. That's perhaps not so surprising, but this example may widen your eyes a bit:

```

class Base { ... };
class Derived: public Base { ... };
Derived d;
Base *pb = &d;           // implicitly convert Derived* => Base*

```

Here we're just creating a base class pointer to a derived class object, but sometimes, the two pointer values will not be the same. When that's the case, an offset is applied *at runtime* to the `Derived*` pointer to get the correct `Base*` pointer value.

This last example demonstrates that a single object (e.g., an object of type `Derived`) might have more than one address (e.g., its address when pointed to by a `Base*` pointer and its address when pointed to by a `Derived*` pointer). That can't happen in C. It can't happen in Java. It can't happen in C#. It *does* happen in C++. In fact, when multiple

inheritance is in use, it happens virtually all the time, but it can happen under single inheritance, too. Among other things, that means you should generally avoid making assumptions about how things are laid out in C++, and you should certainly not perform casts based on such assumptions. For example, casting object addresses to `char*` pointers and then using pointer arithmetic on them almost always yields undefined behavior.

But note that I said that an offset is “sometimes” required. The way objects are laid out and the way their addresses are calculated varies from compiler to compiler. That means that just because your “I know how things are laid out” casts work on one platform doesn’t mean they’ll work on others. The world is filled with woeful programmers who’ve learned this lesson the hard way.

An interesting thing about casts is that it’s easy to write something that looks right (and might be right in other languages) but is wrong. Many application frameworks, for example, require that virtual member function implementations in derived classes call their base class counterparts first. Suppose we have a `Window` base class and a `SpecialWindow` derived class, both of which define the virtual function `onResize`. Further suppose that `SpecialWindow`’s `onResize` is expected to invoke `Window`’s `onResize` first. Here’s a way to implement this that looks like it does the right thing, but doesn’t:

```
class Window {                                // base class
public:
    virtual void onResize() { ... }           // base onResize impl
    ...
};

class SpecialWindow: public Window {          // derived class
public:
    virtual void onResize() {                 // derived onResize impl;
        static_cast<Window>(*this).onResize(); // cast *this to Window,
                                                // then call its onResize;
                                                // this doesn't work!
    }                                         // do SpecialWindow-
                                                // specific stuff
    ...
};
```

I’ve highlighted the cast in the code. (It’s a new-style cast, but using an old-style cast wouldn’t change anything.) As you would expect, the code casts `*this` to a `Window`. The resulting call to `onResize` therefore invokes `Window::onResize`. What you might not expect is that it does not invoke that function on the current object! Instead, the cast cre-

ates a new, temporary *copy* of the base class part of `*this`, then invokes `onResize` on the copy! The above code doesn't call `Window::onResize` on the current object and then perform the `SpecialWindow`-specific actions on that object — it calls `Window::onResize` on a *copy of the base class part* of the current object before performing `SpecialWindow`-specific actions on the current object. If `Window::onResize` modifies the current object (hardly a remote possibility, since `onResize` is a non-const member function), the current object won't be modified. Instead, a *copy* of that object will be modified. If `SpecialWindow::onResize` modifies the current object, however, the current object *will* be modified, leading to the prospect that the code will leave the current object in an invalid state, one where base class modifications have not been made, but derived class ones have been.

The solution is to eliminate the cast, replacing it with what you really want to say. You don't want to trick compilers into treating `*this` as a base class object; you want to call the base class version of `onResize` on the current object. So say that:

```
class SpecialWindow: public Window {
public:
    virtual void onResize() {
        Window::onResize();           // call Window::onResize
        ...                           // on *this
    }
    ...
};
```

This example also demonstrates that if you find yourself wanting to cast, it's a sign that you could be approaching things the wrong way. This is especially the case if your want is for `dynamic_cast`.

Before delving into the design implications of `dynamic_cast`, it's worth observing that many implementations of `dynamic_cast` can be quite slow. For example, at least one common implementation is based in part on string comparisons of class names. If you're performing a `dynamic_cast` on an object in a single-inheritance hierarchy four levels deep, each `dynamic_cast` under such an implementation could cost you up to four calls to `strcmp` to compare class names. A deeper hierarchy or one using multiple inheritance would be more expensive. There are reasons that some implementations work this way (they have to do with support for dynamic linking). Nonetheless, in addition to being leery of casts in general, you should be especially leery of `dynamic_casts` in performance-sensitive code.

The need for `dynamic_cast` generally arises because you want to perform derived class operations on what you believe to be a derived class

object, but you have only a pointer- or reference-to-base through which to manipulate the object. There are two general ways to avoid this problem.

First, use containers that store pointers (often smart pointers — see [Item 13](#)) to derived class objects directly, thus eliminating the need to manipulate such objects through base class interfaces. For example, if, in our `Window/SpecialWindow` hierarchy, only `SpecialWindows` support blinking, instead of doing this:

```
class Window { ... };
class SpecialWindow: public Window {
public:
    void blink();
    ...
};
typedef std::vector<std::tr1::shared_ptr<Window> > VPW; // see Item 13 for info
// on tr1::shared_ptr
VPW winPtrs;
...
for (VPW::iterator iter = winPtrs.begin();           // undesirable code:
     iter != winPtrs.end();                          // uses dynamic_cast
     ++iter) {
    if (SpecialWindow *psw = dynamic_cast<SpecialWindow*>(iter->get()))
        psw->blink();
}
```

try to do this instead:

```
typedef std::vector<std::tr1::shared_ptr<SpecialWindow> > VPSW;
VPSW winPtrs;
...
for (VPSW::iterator iter = winPtrs.begin();          // better code: uses
     iter != winPtrs.end();                          // no dynamic_cast
     ++iter)
    (*iter)->blink();
```

Of course, this approach won't allow you to store pointers to all possible `Window` derivatives in the same container. To work with different window types, you might need multiple type-safe containers.

An alternative that will let you manipulate all possible `Window` derivatives through a base class interface is to provide virtual functions in the base class that let you do what you need. For example, though only `SpecialWindows` can blink, maybe it makes sense to declare the

function in the base class, offering a default implementation that does nothing:

```
class Window {
public:
    virtual void blink() {}           // default impl is no-op;
    ...                               // see Item 34 for why
};                                   // a default impl may be
                                   // a bad idea

class SpecialWindow: public Window {
public:
    virtual void blink() { ... }     // in this class, blink
    ...                               // does something
};

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;
VPW winPtrs;                         // container holds
                                   // (ptrs to) all possible
...                                   // Window types

for (VPW::iterator iter = winPtrs.begin();
     iter != winPtrs.end();
     ++iter)                          // note lack of
    (*iter)->blink();                 // dynamic_cast
```

Neither of these approaches — using type-safe containers or moving virtual functions up the hierarchy — is universally applicable, but in many cases, they provide a viable alternative to `dynamic_casting`. When they do, you should embrace them.

One thing you definitely want to avoid is designs that involve cascading `dynamic_casts`, i.e., anything that looks like this:

```
class Window { ... };
...
// derived classes are defined here
typedef std::vector<std::tr1::shared_ptr<Window> > VPW;
VPW winPtrs;
...
for (VPW::iterator iter = winPtrs.begin(); iter != winPtrs.end(); ++iter)
{
    if (SpecialWindow1 *psw1 =
        dynamic_cast<SpecialWindow1*>(iter->get())) { ... }
    else if (SpecialWindow2 *psw2 =
        dynamic_cast<SpecialWindow2*>(iter->get())) { ... }
    else if (SpecialWindow3 *psw3 =
        dynamic_cast<SpecialWindow3*>(iter->get())) { ... }
    ...
}
```

Such C++ generates code that's big and slow, plus it's brittle, because every time the Window class hierarchy changes, all such code has to be examined to see if it needs to be updated. (For example, if a new derived class gets added, a new conditional branch probably needs to be added to the above cascade.) Code that looks like this should almost always be replaced with something based on virtual function calls.

Good C++ uses very few casts, but it's generally not practical to get rid of all of them. The cast from int to double on [page 118](#), for example, is a reasonable use of a cast, though it's not strictly necessary. (The code could be rewritten to declare a new variable of type double that's initialized with x's value.) Like most suspicious constructs, casts should be isolated as much as possible, typically hidden inside functions whose interfaces shield callers from the grubby work being done inside.

Things to Remember

- ◆ Avoid casts whenever practical, especially `dynamic_casts` in performance-sensitive code. If a design requires casting, try to develop a cast-free alternative.
- ◆ When casting is necessary, try to hide it inside a function. Clients can then call the function instead of putting casts in their own code.
- ◆ Prefer C++-style casts to old-style casts. They are easier to see, and they are more specific about what they do.

Item 28: Avoid returning “handles” to object internals.

Suppose you're working on an application involving rectangles. Each rectangle can be represented by its upper left corner and its lower right corner. To keep a Rectangle object small, you might decide that the points defining its extent shouldn't be stored in the Rectangle itself, but rather in an auxiliary struct that the Rectangle points to:

```
class Point { // class for representing points
public:
    Point(int x, int y);
    ...
    void setX(int newVal);
    void setY(int newVal);
    ...
};
```

```

template<typename T>                                // declare
const Rational<T> doMultiply( const Rational<T>& lhs, // helper
                              const Rational<T>& rhs); // template

template<typename T>
class Rational {
public:
    ...
friend
    const Rational<T> operator*(const Rational<T>& lhs,
                                const Rational<T>& rhs) // Have friend
    { return doMultiply(lhs, rhs); } // call helper
    ...
};

```

Many compilers essentially force you to put all template definitions in header files, so you may need to define `doMultiply` in your header as well. (As [Item 30](#) explains, such templates need not be inline.) That could look like this:

```

template<typename T>                                // define
const Rational<T> doMultiply(const Rational<T>& lhs, // helper
                              const Rational<T>& rhs) // template in
{                                                                    // header file,
    return Rational<T>(lhs.numerator() * rhs.numerator(), // if necessary
                      lhs.denominator() * rhs.denominator());
}

```

As a template, of course, `doMultiply` won't support mixed-mode multiplication, but it doesn't need to. It will only be called by `operator*`, and `operator*` *does* support mixed-mode operations! In essence, the *function* `operator*` supports whatever type conversions are necessary to ensure that two `Rational` objects are being multiplied, then it passes these two objects to an appropriate instantiation of the `doMultiply` *template* to do the actual multiplication. Synergy in action, no?

Things to Remember

- ◆ When writing a class template that offers functions related to the template that support implicit type conversions on all parameters, define those functions as friends inside the class template.

Item 47: Use traits classes for information about types.

The STL is primarily made up of templates for containers, iterators, and algorithms, but it also has a few utility templates. One of these is called `advance`. `advance` moves a specified iterator a specified distance:

```
template<typename IterT, typename DistT> // move iter d units
void advance(IterT& iter, DistT d); // forward; if d < 0,
// move iter backward
```

Conceptually, `advance` just does `iter += d`, but `advance` can't be implemented that way, because only random access iterators support the `+=` operation. Less powerful iterator types have to implement `advance` by iteratively applying `++` or `--` `d` times.

Um, you don't remember your STL iterator categories? No problem, we'll do a mini-review. There are five categories of iterators, corresponding to the operations they support. *Input iterators* can move only forward, can move only one step at a time, can only read what they point to, and can read what they're pointing to only once. They're modeled on the read pointer into an input file; the C++ library's `istream_iterators` are representative of this category. *Output iterators* are analogous, but for output: they move only forward, move only one step at a time, can only write what they point to, and can write it only once. They're modeled on the write pointer into an output file; `ostream_iterators` epitomize this category. These are the two least powerful iterator categories. Because input and output iterators can move only forward and can read or write what they point to at most once, they are suitable only for one-pass algorithms.

A more powerful iterator category consists of *forward iterators*. Such iterators can do everything input and output iterators can do, plus they can read or write what they point to more than once. This makes them viable for multi-pass algorithms. The STL offers no singly linked list, but some libraries offer one (usually called `slist`), and iterators into such containers are forward iterators. Iterators into TR1's hashed containers (see [Item 54](#)) may also be in the forward category.

Bidirectional iterators add to forward iterators the ability to move backward as well as forward. Iterators for the STL's `list` are in this category, as are iterators for `set`, `multiset`, `map`, and `multimap`.

The most powerful iterator category is that of *random access iterators*. These kinds of iterators add to bidirectional iterators the ability to perform "iterator arithmetic," i.e., to jump forward or backward an arbitrary distance in constant time. Such arithmetic is analogous to pointer arithmetic, which is not surprising, because random access iterators are modeled on built-in pointers, and built-in pointers can act as random access iterators. Iterators for `vector`, `deque`, and `string` are random access iterators.

For each of the five iterator categories, C++ has a "tag struct" in the standard library that serves to identify it:

```

struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};

```

The inheritance relationships among these structs are valid is-a relationships (see [Item 32](#)): it's true that all forward iterators are also input iterators, etc. We'll see the utility of this inheritance shortly.

But back to `advance`. Given the different iterator capabilities, one way to implement `advance` would be to use the lowest-common-denominator strategy of a loop that iteratively increments or decrements the iterator. However, that approach would take linear time. Random access iterators support constant-time iterator arithmetic, and we'd like to take advantage of that ability when it's present.

What we really want to do is implement `advance` essentially like this:

```

template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
    if (iter is a random access iterator) {
        iter += d; // use iterator arithmetic
                // for random access iters
    }
    else {
        if (d >= 0) { while (d--) ++iter; } // use iterative calls to
        else { while (d++) --iter; }      // ++ or -- for other
    } // iterator categories
}

```

This requires being able to determine whether `iter` is a random access iterator, which in turn requires knowing whether its type, `IterT`, is a random access iterator type. In other words, we need to get some information about a type. That's what *traits* let you do: they allow you to get information about a type during compilation.

Traits aren't a keyword or a predefined construct in C++; they're a technique and a convention followed by C++ programmers. One of the demands made on the technique is that it has to work as well for built-in types as it does for user-defined types. For example, if `advance` is called with a pointer (like a `const char*`) and an `int`, `advance` has to work, but that means that the traits technique must apply to built-in types like pointers.

The fact that traits must work with built-in types means that things like nesting information inside types won't do, because there's no way to nest information inside pointers. The traits information for a type, then, must be external to the type. The standard technique is to put it

into a template and one or more specializations of that template. For iterators, the template in the standard library is named `iterator_traits`:

```
template<typename IterT>           // template for information about
struct iterator_traits;           // iterator types
```

As you can see, `iterator_traits` is a struct. By convention, traits are always implemented as structs. Another convention is that the structs used to implement traits are known as — I am not making this up — traits *classes*.

The way `iterator_traits` works is that for each type `IterT`, a typedef named `iterator_category` is declared in the struct `iterator_traits<IterT>`. This typedef identifies the iterator category of `IterT`.

`iterator_traits` implements this in two parts. First, it imposes the requirement that any user-defined iterator type must contain a nested typedef named `iterator_category` that identifies the appropriate tag struct. `deque`'s iterators are random access, for example, so a class for `deque` iterators would look something like this:

```
template < ... >                 // template params elided
class deque {
public:
    class iterator {
    public:
        typedef random_access_iterator_tag iterator_category;
        ...
    };
    ...
};
```

`list`'s iterators are bidirectional, however, so they'd do things this way:

```
template < ... >
class list {
public:
    class iterator {
    public:
        typedef bidirectional_iterator_tag iterator_category;
        ...
    };
    ...
};
```

`iterator_traits` just parrots back the iterator class's nested typedef:

```
// the iterator_category for type IterT is whatever IterT says it is;
// see Item 42 for info on the use of "typedef typename"
template<typename IterT>
struct iterator_traits {
    typedef typename IterT::iterator_category iterator_category;
    ...
};
```

This works well for user-defined types, but it doesn't work at all for iterators that are pointers, because there's no such thing as a pointer with a nested typedef. The second part of the `iterator_traits` implementation handles iterators that are pointers.

To support such iterators, `iterator_traits` offers a *partial template specialization* for pointer types. Pointers act as random access iterators, so that's the category `iterator_traits` specifies for them:

```
template<typename T>                // partial template specialization
struct iterator_traits<T*>          // for built-in pointer types
{
    typedef random_access_iterator_tag iterator_category;
    ...
};
```

At this point, you know how to design and implement a traits class:

- Identify some information about types you'd like to make available (e.g., for iterators, their iterator category).
- Choose a name to identify that information (e.g., `iterator_category`).
- Provide a template and set of specializations (e.g., `iterator_traits`) that contain the information for the types you want to support.

Given `iterator_traits` — actually `std::iterator_traits`, since it's part of C++'s standard library — we can refine our pseudocode for advance:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
    if (typeid(typename std::iterator_traits<IterT>::iterator_category) ==
        typeid(std::random_access_iterator_tag))
    ...
}
```

Although this looks promising, it's not what we want. For one thing, it will lead to compilation problems, but we'll explore that in [Item 48](#); right now, there's a more fundamental issue to consider. `IterT`'s type is known during compilation, so `iterator_traits<IterT>::iterator_category` can also be determined during compilation. Yet the `if` statement is evaluated at runtime (unless your optimizer is crafty enough to get rid of it). Why do something at runtime that we can do during compilation? It wastes time (literally), and it bloats our executable.

What we really want is a conditional construct (i.e., an `if...else` statement) for types that is evaluated during compilation. As it happens, C++ already has a way to get that behavior. It's called overloading.

When you overload some function `f`, you specify different parameter types for the different overloads. When you call `f`, compilers pick the

best overload, based on the arguments you're passing. Compilers essentially say, "If this overload is the best match for what's being passed, call this `f`; if this other overload is the best match, call it; if this third one is best, call it," etc. See? A compile-time conditional construct for types. To get advance to behave the way we want, all we have to do is create multiple versions of an overloaded function containing the "guts" of advance, declaring each to take a different type of `iterator_category` object. I use the name `doAdvance` for these functions:

```

template<typename IterT, typename DistT>           // use this impl for
void doAdvance(IterT& iter, DistT d,             // random access
               std::random_access_iterator_tag)  // iterators
{
    iter += d;
}

template<typename IterT, typename DistT>         // use this impl for
void doAdvance(IterT& iter, DistT d,             // bidirectional
               std::bidirectional_iterator_tag)  // iterators
{
    if (d >= 0) { while (d--) ++iter; }
    else { while (d++) --iter; }
}

template<typename IterT, typename DistT>         // use this impl for
void doAdvance(IterT& iter, DistT d,             // input iterators
               std::input_iterator_tag)
{
    if (d < 0) {
        throw std::out_of_range("Negative distance"); // see below
    }
    while (d--) ++iter;
}

```

Because `forward_iterator_tag` inherits from `input_iterator_tag`, the version of `doAdvance` for `input_iterator_tag` will also handle forward iterators. That's the motivation for inheritance among the various `iterator_tag` structs. (In fact, it's part of the motivation for *all* public inheritance: to be able to write code for base class types that also works for derived class types.)

The specification for `advance` allows both positive and negative distances for random access and bidirectional iterators, but behavior is undefined if you try to move a forward or input iterator a negative distance. The implementations I checked simply assumed that `d` was non-negative, thus entering a *very* long loop counting "down" to zero if a negative distance was passed in. In the code above, I've shown an exception being thrown instead. Both implementations are valid. That's the curse of undefined behavior: you *can't predict* what will happen.

Given the various overloads for `doAdvance`, all `advance` needs to do is call them, passing an extra object of the appropriate iterator category type so that the compiler will use overloading resolution to call the proper implementation:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
    doAdvance(                                // call the version
        iter, d,                               // of doAdvance
        typename                               // that is
            std::iterator_traits<IterT>::iterator_category() // appropriate for
    );                                          // iter's iterator
}                                              // category
```

We can now summarize how to use a traits class:

- Create a set of overloaded “worker” functions or function templates (e.g., `doAdvance`) that differ in a traits parameter. Implement each function in accord with the traits information passed.
- Create a “master” function or function template (e.g., `advance`) that calls the workers, passing information provided by a traits class.

Traits are widely used in the standard library. There’s `iterator_traits`, of course, which, in addition to `iterator_category`, offers four other pieces of information about iterators (the most useful of which is `value_type` — [Item 42](#) shows an example of its use). There’s also `char_traits`, which holds information about character types, and `numeric_limits`, which serves up information about numeric types, e.g., their minimum and maximum representable values, etc. (The name `numeric_limits` is a bit of a surprise, because the more common convention is for traits classes to end with “traits,” but `numeric_limits` is what it’s called, so `numeric_limits` is the name we use.)

TR1 (see [Item 54](#)) introduces a slew of new traits classes that give information about types, including `is_fundamental<T>` (whether `T` is a built-in type), `is_array<T>` (whether `T` is an array type), and `is_base_of<T1, T2>` (whether `T1` is the same as or is a base class of `T2`). All told, TR1 adds over 50 traits classes to standard C++.

Things to Remember

- ♦ Traits classes make information about types available during compilation. They’re implemented using templates and template specializations.
- ♦ In conjunction with overloading, traits classes make it possible to perform compile-time `if...else` tests on types.

Index

Operators are listed under *operator*. That is, `operator<<` is listed under `operator<<`, not under `<<`, etc.

Example classes, structs, and class or struct templates are indexed under *example classes/templates*. Example function and function templates are indexed under *example functions/templates*.

Before A

.NET 7, 81, 135, 145, 194
see also C#
=, in initialization vs. assignment 6
1066 150
2nd edition of this book
 compared to 3rd edition xv–xvi, 277–279
 see also inside back cover
3rd edition of this book
 compared to 2nd edition xv–xvi, 277–279
 see also inside back cover
80-20 rule 139, 168

A

Abrahams, David xvii, xviii, xix
abstract classes 43
accessibility
 control over data members' 95
 name, multiple inheritance and 193
accessing names, in templated
 bases 207–212
addresses
 inline functions 136
 objects 118
aggregation, see *composition*
Alexandrescu, Andrei xix
aliasing 54
alignment 249–250
allocators, in the STL 240

alternatives to virtual functions 169–177
ambiguity
 multiple inheritance and 192
 nested dependent names and types 205
Arbiter, Petronius vii
argument-dependent lookup 110
arithmetic, mixed-mode 103, 222–226
array layout, vs. object layout 73
array new 254–255
array, invalid index and 7
ASPECT_RATIO 13
assignment
 see also `operator=`
 chaining assignments 52
 copy-and-swap and 56
 generalized 220
 to self, `operator=` and 53–57
 vs. initialization 6, 27–29, 114
assignment operator, copy 5
auto_ptr, see `std::auto_ptr`
automatically generated functions 34–37
 copy constructor and copy assignment
 operator 221
 disallowing 37–39
avoiding code duplication 50, 60

B

Bai, Yun xix
Barry, Dave, allusion to 229
Bartolucci, Guido xix

- base classes
 - copying 59
 - duplication of data in 193
 - lookup in, this-> and 210
 - names hidden in derived classes 263
 - polymorphic 44
 - polymorphic, destructors and 40–44
 - templated 207–212
 - virtual 193
 - basic guarantee, the 128
 - Battle of Hastings 150
 - Berck, Benjamin xix
 - bidirectional iterators 227
 - bidirectional_iterator_tag 228
 - binary upgradeability, inlining and 138
 - binding
 - dynamic, see [dynamic binding](#)
 - static, see [static binding](#)
 - birds and penguins 151–153
 - bitwise const member functions 21–22
 - books
 - C++ Programming Language, The* xvii
 - C++ Templates* xviii
 - Design Patterns* xvii
 - Effective STL* 273, 275–276
 - Exceptional C++* xvii
 - Exceptional C++ Style* xvii, xviii
 - More Effective C++* 273, 273–274
 - More Exceptional C++* xvii
 - Satyricon* vii
 - Some Must Watch While Some Must Sleep* 150
 - Boost 10, 269–272
 - containers 271
 - Conversion library 270
 - correctness and testing support 272
 - data structures 272
 - function objects and higher-order programming utilities 271
 - functionality not provided 272
 - generic programming support 271
 - Graph library 270
 - inter-language support 272
 - Lambda library 271
 - math and numerics utilities 271
 - memory management utilities 272
 - MPL library 270, 271
 - noncopyable base class 39
 - Pool library 250, 251
 - scoped_array 65, 216, 272
 - shared_array 65
 - shared_ptr implementation, costs 83
 - smart pointers 65, 272
 - web page xvii
 - string and text utilities 271
 - template metaprogramming support 271
 - TR1 and 9–10, 268, 269
 - typelist support 271
 - web site 10, 269, 272
 - boost, as synonym for std::tr1 268
 - Bosch, Derek xviii
 - breakpoints, and inlining 139
 - Buffy the Vampire Slayer* xx
 - bugs, reporting xvi
 - built-in types 26–27
 - efficiency and passing 89
 - incompatibilities with 80
- ## C
- C standard library and C++ standard library 264
 - C# 43, 76, 97, 100, 116, 118, 190
 - see also [.NET](#)
 - C++ Programming Language, The* xvii
 - C++ standard library 263–269
 - <iosfwd> and 144
 - array replacements and 75
 - C standard library and 264
 - C89 standard library and 264
 - header organization of 101
 - list template 186
 - logic_error and 113
 - set template 185
 - vector template 75
 - C++ Templates* xviii
 - C++, as language federation 11–13
 - C++0x 264
 - C++-style casts 117
 - C, as sublanguage of C++ 12
 - C99 standard library, TR1 and 267
 - caching
 - const and 22
 - mutable and 22
 - Cai, Steve xix
 - calling swap 110
 - calls to base classes, casting and 119
 - Cargill, Tom xviii
 - Carrara, Enrico xix
 - Carroll, Glenn xviii
 - casting 116–123
 - see also [const_cast](#), [static_cast](#), [dynamic_cast](#), and [reinterpret_cast](#)
 - base class calls and 119
 - constness away 24–25
 - encapsulation and 123
 - grep and 117
 - syntactic forms 116–117
 - type systems and 116
 - undefined behavior and 119
 - chaining assignments 52

- Chang, Brandon [xix](#)
- Clamage, Steve [xviii](#)
- class definitions
 - artificial client dependencies, eliminating [143](#)
 - class declarations vs. [143](#)
 - object sizes and [141](#)
- class design, see [type design](#)
- class names, explicitly specifying [162](#)
- class, vs. typename [203](#)
- classes
 - see also [class definitions](#), [interfaces](#)
 - abstract [43](#), [162](#)
 - base
 - see also [base classes](#)
 - duplication of data in [193](#)
 - polymorphic [44](#)
 - templated [207–212](#)
 - virtual [193](#)
 - defining [4](#)
 - derived
 - see also [inheritance](#)
 - virtual base initialization of [194](#)
 - Handle [144–145](#)
 - Interface [145–147](#)
 - meaning of no virtual functions [41](#)
 - RAII, see [RAII](#)
 - specification, see [interfaces](#)
 - traits [226–232](#)
- client [7](#)
- clustering objects [251](#)
- code
 - bloat [24](#), [135](#), [230](#)
 - avoiding, in templates [212–217](#)
 - copy assignment operator [60](#)
 - duplication, see [duplication](#)
 - exception-safe [127–134](#)
 - factoring out of templates [212–217](#)
 - incorrect, efficiency and [90](#)
 - reuse [195](#)
 - sharing, see [duplication](#), [avoiding](#)
- Cohen, Jake [xix](#)
- Comeau, Greg [xviii](#)
 - URL for his C/C++ FAQ [xviii](#)
- common features and inheritance [164](#)
- commonality and variability analysis [212](#)
- compatibility, vptrs and [42](#)
- compatible types, accepting [218–222](#)
- compilation dependencies [140–148](#)
 - minimizing [140–148](#), [190](#)
 - pointers, references, and objects and [143](#)
- compiler warnings [262–263](#)
 - calls to virtuals and [50](#)
 - inlining and [136](#)
 - partial copies and [58](#)
- compiler-generated functions [34–37](#)
 - disallowing [37–39](#)
 - functions compilers may generate [221](#)
- compilers
 - parsing nested dependent names [204](#)
 - programs executing within, see [template metaprogramming](#)
 - register usage and [89](#)
 - reordering operations [76](#)
 - typename and [207](#)
 - when errors are diagnosed [212](#)
- compile-time polymorphism [201](#)
- composition [184–186](#)
 - meanings of [184](#)
 - replacing private inheritance with [189](#)
 - synonyms for [184](#)
 - vs. private inheritance [188](#)
- conceptual constness, see [const](#), [logical](#)
- consistency with the built-in types [19](#), [86](#)
- const [13](#), [17–26](#)
 - bitwise [21–22](#)
 - caching and [22](#)
 - casting away [24–25](#)
 - function declarations and [18](#)
 - logical [22–23](#)
 - member functions [19–25](#)
 - duplication and [23–25](#)
 - members, initialization of [29](#)
 - overloading on [19–20](#)
 - pass by reference and [86–90](#)
 - passing `std::auto_ptr` and [220](#)
 - pointers [17](#)
 - return value [18](#)
 - uses [17](#)
 - vs. `#define` [13–14](#)
- `const_cast` [25](#), [117](#)
 - see also [casting](#)
- `const_iterator`, vs. `iterator` [18](#)
- constants, see [const](#)
- constraints on interfaces, from inheritance [85](#)
- constructors [84](#)
 - copy [5](#)
 - default [4](#)
 - empty, illusion of [137](#)
 - explicit [5](#), [85](#), [104](#)
 - implicitly generated [34](#)
 - inlining and [137–138](#)
 - operator new and [137](#)
 - possible implementation in derived classes [138](#)
 - relationship to new [73](#)
 - static functions and [52](#)
 - virtual [146](#), [147](#)
 - virtual functions and [48–52](#)
 - with vs. without arguments [114](#)
- containers, in Boost [271](#)

containment, see [composition](#)
 continue, delete and [62](#)
 control over data members'
 accessibility [95](#)
 convenience functions [100](#)
 Conversion library, in Boost [270](#)
 conversions, type, see [type conversions](#)
 copies, partial [58](#)
 copy assignment operator [5](#)
 code in copy constructor and [60](#)
 derived classes and [60](#)
 copy constructors
 default definition [35](#)
 derived classes and [60](#)
 generalized [219](#)
 how used [5](#)
 implicitly generated [34](#)
 pass-by-value and [6](#)
 copy-and-swap [131](#)
 assignment and [56](#)
 exception-safe code and [132](#)
 copying
 base class parts [59](#)
 behavior, resource management
 and [66-69](#)
 functions, the [57](#)
 objects [57-60](#)
 correctness
 designing interfaces for [78-83](#)
 testing and, Boost support [272](#)
 corresponding forms of new and
 delete [73-75](#)
 corrupt data structures, exception-safe
 code and [127](#)
 cows, coming home [139](#)
 crimes against English [39, 204](#)
 cross-DLL problem [82](#)
 CRTP [246](#)
 C-style casts [116](#)
 ctor [8](#)
 curiously recurring template pattern [246](#)

D

dangling handles [126](#)
 Dashtinezhad, Sasan [xix](#)
 data members
 adding, copying functions and [58](#)
 control over accessibility [95](#)
 protected [97](#)
 static, initialization of [242](#)
 why private [94-98](#)
 data structures
 exception-safe code and [127](#)
 in Boost [272](#)
 Davis, Tony [xviii](#)

deadly MI diamond [193](#)
 debuggers
 #define and [13](#)
 inline functions and [139](#)
 declarations [3](#)
 inline functions [135](#)
 replacing definitions [143](#)
 static const integral members [14](#)
 default constructors [4](#)
 construction with arguments vs. [114](#)
 implicitly generated [34](#)
 default implementations
 for virtual functions, danger of [163-167](#)
 of copy constructor [35](#)
 of operator= [35](#)
 default initialization, unintended [59](#)
 default parameters [180-183](#)
 impact if changed [183](#)
 static binding of [182](#)
 #define
 debuggers and [13](#)
 disadvantages of [13, 16](#)
 vs. const [13-14](#)
 vs. inline functions [16-17](#)
 definitions [4](#)
 classes [4](#)
 deliberate omission of [38](#)
 functions [4](#)
 implicitly generated functions [35](#)
 objects [4](#)
 pure virtual functions [162, 166-167](#)
 replacing with declarations [143](#)
 static class members [242](#)
 static const integral members [14](#)
 templates [4](#)
 variable, postponing [113-116](#)
 delete
 see also [operator delete](#)
 forms of [73-75](#)
 operator delete and [73](#)
 relationship to destructors [73](#)
 usage problem scenarios [62](#)
 delete [], std::auto_ptr and tr1::shared_ptr
 and [65](#)
 deleters
 std::auto_ptr and [68](#)
 tr1::shared_ptr and [68, 81-83](#)
 Delphi [97](#)
 Dement, William [150](#)
 dependencies, compilation [140-148](#)
 dependent names [204](#)
 dereferencing a null pointer, undefined
 behavior of [6](#)
 derived classes
 copy assignment operators and [60](#)
 copy constructors and [60](#)
 hiding names in base classes [263](#)

- implementing constructors in 138
- virtual base initialization and 194
- design
 - contradiction in 179
 - of interfaces 78–83
 - of types 78–86
- Design Patterns* xvii
- design patterns
 - curiously recurring template (CRTP) 246
 - encapsulation and 173
 - generating from templates 237
 - Singleton 31
 - Strategy 171–177
 - Template Method 170
 - TMP and 237
- destructors 84
 - exceptions and 44–48
 - inlining and 137–138
 - pure virtual 43
 - relationship to delete 73
 - resource managing objects and 63
 - static functions and 52
 - virtual
 - operator delete and 255
 - polymorphic base classes and 40–44
 - virtual functions and 48–52
- Dewhurst, Steve xvii
- dimensional unit correctness, TMP and 236
- DLLs, delete and 82
- dtor 8
- Dulimov, Peter xix
- duplication
 - avoiding 23–25, 29, 50, 60, 164, 183, 212–217
 - base class data and 193
 - init function and 60
- dynamic binding
 - definition of 181
 - of virtual functions 179
- dynamic type, definition of 181
- dynamic_cast 50, 117, 120–123
 - see also [casting](#)
 - efficiency of 120

E

- early binding 180
- easy to use correctly and hard to use incorrectly 78–83
- EBO, see [empty base optimization](#)
- Effective C++*, compared to *More Effective C++* and *Effective STL* 273
- Effective STL* 273, 275–276
 - compared to *Effective C++* 273

- contents of 275–276
- efficiency
 - assignment vs. construction and destruction 94
 - default parameter binding 182
 - dynamic_cast 120
 - Handle classes 147
 - incorrect code and 90, 94
 - init. with vs. without args 114
 - Interface classes 147
 - macros vs. inline functions 16
 - member init. vs. assignment 28
 - minimizing compilation dependencies 147
 - operator new/operator delete and 248
 - pass-by-reference and 87
 - pass-by-value and 86–87
 - passing built-in types and 89
 - runtime vs. compile-time tests 230
 - template metaprogramming and 233
 - template vs. function parameters 216
 - unused objects 113
 - virtual functions 168
- Eiffel 100
- embedding, see [composition](#)
- empty base optimization (EBO) 190–191
- encapsulation 95, 99
 - casts and 123
 - design patterns and 173
 - handles and 125
 - measuring 99
 - protected members and 97
 - RAII classes and 72
- enum hack 15–16, 236
- errata list, for this book xvi
- errors
 - detected during linking 39, 44
 - runtime 152
- evaluation order, of parameters 76
- example classes/templates
 - A 4
 - ABEntry 27
 - AccessLevels 95
 - Address 184
 - Airplane 164, 165, 166
 - Airport 164
 - AtomicClock 40
 - AWOV 43
 - B 4, 178, 262
 - Base 54, 118, 137, 157, 158, 159, 160, 254, 255, 259
 - BelowBottom 219
 - bidirectional_iterator_tag 228
 - Bird 151, 152, 153
 - Bitmap 54
 - BorrowableItem 192
 - Bottom 218
 - BuyTransaction 49, 51

- C 5
- Circle 181
- CompanyA 208
- CompanyB 208
- CompanyZ 209
- CostEstimate 15
- CPerson 198
- CTextBlock 21, 22, 23
- Customer 57, 58
- D 178, 262
- DatabaseID 197
- Date 58, 79
- Day 79
- DBConn 45, 47
- DBConnection 45
- deque 229
- deque::iterator 229
- Derived 54, 118, 137, 157, 158, 159, 160, 206, 254, 260
- Directory 31
- ElectronicGadget 192
- Ellipse 161
- Empty 34, 190
- EvilBadGuy 172, 174
- EyeCandyCharacter 175
- Factorial 235
- Factorial<0> 235
- File 193, 194
- FileSystem 30
- FlyingBird 152
- Font 71
- forward_iterator_tag 228
- GameCharacter 169, 170, 172, 173, 176
- GameLevel 174
- GamePlayer 14, 15
- GraphNode 4
- GUIObject 126
- HealthCalcFunc 176
- HealthCalculator 174
- HoldsAnInt 190, 191
- HomeForSale 37, 38, 39
- input_iterator_tag 228
- input_iterator_tag<lter*> 230
- InputFile 193, 194
- Investment 61, 70
- IOFile 193, 194
- IPerson 195, 197
- iterator_traits 229
 - see also *std::iterator_traits*
- list 229
- list::iterator 229
- Lock 66, 67, 68
- LoggingMsgSender 208, 210, 211
- Middle 218
- ModelA 164, 165, 167
- ModelB 164, 165, 167
- ModelC 164, 166, 167
- Month 79, 80
- MP3Player 192
- MsgInfo 208
- MsgSender 208
- MsgSender<CompanyZ> 209
- NamedObject 35, 36
- NewHandlerHolder 243
- NewHandlerSupport 245
- output_iterator_tag 228
- OutputFile 193, 194
- Penguin 151, 152, 153
- Person 86, 135, 140, 141, 142, 145, 146, 150, 184, 187
- PersonInfo 195, 197
- PhoneNumber 27, 184
- PMImpl 131
- Point 26, 41, 123
- PrettyMenu 127, 130, 131
- PriorityCustomer 58
- random_access_iterator_tag 228
- Rational 90, 102, 103, 105, 222, 223, 224, 225, 226
- RealPerson 147
- Rectangle 124, 125, 154, 161, 181, 183
- RectData 124
- SellTransaction 49
- Set 185
- Shape 161, 162, 163, 167, 180, 182, 183
- SmartPtr 218, 219, 220
- SpecialString 42
- SpecialWindow 119, 120, 121, 122
- SpeedDataCollection 96
- Square 154
- SquareMatrix 213, 214, 215, 216
- SquareMatrixBase 214, 215
- StandardNewDeleteForms 260
- Student 86, 150, 187
- TextBlock 20, 23, 24
- TimeKeeper 40, 41
- Timer 188
- Top 218
- Transaction 48, 50, 51
- Uncopyable 39
- WaterClock 40
- WebBrowser 98, 100, 101
- Widget 4, 5, 44, 52, 53, 54, 56, 107, 108, 109, 118, 189, 199, 201, 242, 245, 246, 257, 258, 261
- Widget::WidgetTimer 189
- WidgetImpl 106, 108
- Window 88, 119, 121, 122
- WindowWithScrollBars 88
- WristWatch 40
- X 242
- Y 242
- Year 79
- example functions/templates
 - ABEntry::ABEntry 27, 28
 - AccessLevels::getReadOnly 95
 - AccessLevels::getReadWrite 95
 - AccessLevels::setReadOnly 95

- AccessLevels::setWriteOnly 95
- advance 228, 230, 232, 233, 234
- Airplane::defaultFly 165
- Airplane::fly 164, 165, 166, 167
- askUserForDatabaseID 195
- AWOV::AWOV 43
- B::mf 178
- Base::operator delete 255
- Base::operator new 254
- Bird::fly 151
- BorrowableItem::checkOut 192
- boundingBox 126
- BuyTransaction::BuyTransaction 51
- BuyTransaction::createLogString 51
- calcHealth 174
- callWithMax 16
- changeFontSize 71
- Circle::draw 181
- clearAppointments 143, 144
- clearBrowser 98
- CPerson::birthDate 198
- CPerson::CPerson 198
- CPerson::name 198
- CPerson::valueDelimClose 198
- CPerson::valueDelimOpen 198
- createInvestment 62, 70, 81, 82, 83
- CTextBlock::length 22, 23
- CTextBlock::operator[] 21
- Customer::Customer 58
- Customer::operator= 58
- D::mf 178
- Date::Date 79
- Day::Day 79
- daysHeld 69
- DBConn::~DBConn 45, 46, 47
- DBConn::close 47
- defaultHealthCalc 172, 173
- Derived::Derived 138, 206
- Derived::mf1 160
- Derived::mf4 157
- Directory::Directory 31, 32
- doAdvance 231
- doMultiply 226
- doProcessing 200, 202
- doSomething 5, 44, 54, 110
- doSomeWork 118
- eat 151, 187
- ElectronicGadget::checkOut 192
- Empty::~Empty 34
- Empty::Empty 34
- Empty::operator= 34
- encryptPassword 114, 115
- error 152
- EvilBadGuy::EvilBadGuy 172
- f 62, 63, 64
- FlyingBird::fly 152
- Font::~Font 71
- Font::Font 71
- Font::get 71
- Font::operator FontHandle 71
- GameCharacter::doHealthValue 170
- GameCharacter::GameCharacter 172, 174, 176
- GameCharacter::healthValue 169, 170, 172, 174, 176
- GameLevel::health 174
- getFont 70
- hasAcceptableQuality 6
- HealthCalcFunc::calc 176
- HealthCalculator::operator() 174
- lock 66
- Lock::~Lock 66
- Lock::Lock 66, 68
- logCall 57
- LoggingMsgSender::sendClear 208, 210, 211
- loseHealthQuickly 172
- loseHealthSlowly 172
- main 141, 142, 236, 241
- makeBigger 154
- makePerson 195
- max 135
- ModelA::fly 165, 167
- ModelB::fly 165, 167
- ModelC::fly 166, 167
- Month::Dec 80
- Month::Feb 80
- Month::Jan 80
- Month::Month 79, 80
- MsgSender::sendClear 208
- MsgSender::sendSecret 208
- MsgSender<CompanyZ>::sendSecret 209
- NewHandlerHolder::~NewHandlerHolder 243
- NewHandlerHolder::NewHandlerHolder 243
- NewHandlerSupport::operator new 245
- NewHandlerSupport::set_new_handler 245
- numDigits 4
- operator delete 255
- operator new 249, 252
- operator* 91, 92, 94, 105, 222, 224, 225, 226
- operator== 93
- outOfMem 240
- Penguin::fly 152
- Person::age 135
- Person::create 146, 147
- Person::name 145
- Person::Person 145
- PersonInfo::theName 196
- PersonInfo::valueDelimClose 196
- PersonInfo::valueDelimOpen 196
- PrettyMenu::changeBackground 127, 128, 130, 131
- print 20
- print2nd 204, 205
- printNameAndDisplay 88, 89
- priority 75
- PriorityCustomer::operator= 59

PriorityCustomer::PriorityCustomer 59
 processWidget 75
 RealPerson::~~RealPerson 147
 RealPerson::RealPerson 147
 Rectangle::doDraw 183
 Rectangle::draw 181, 183
 Rectangle::lowerRight 124, 125
 Rectangle::upperLeft 124, 125
 releaseFont 70
 Set::insert 186
 Set::member 186
 Set::remove 186
 Set::size 186
 Shape::doDraw 183
 Shape::draw 161, 162, 180, 182, 183
 Shape::error 161, 163
 Shape::objectID 161, 167
 SmartPtr::get 220
 SmartPtr::SmartPtr 220
 someFunc 132, 156
 SpecialWindow::blink 122
 SpecialWindow::onResize 119, 120
 SquareMatrix::invert 214
 SquareMatrix::setDataPtr 215
 SquareMatrix::SquareMatrix 215, 216
 StandardNewDeleteForms::operator delete 260, 261
 StandardNewDeleteForms::operator new 260, 261
 std::swap 109
 std::swap<Widget> 107, 108
 study 151, 187
 swap 106, 109
 tempDir 32
 TextBlock::operator[] 20, 23, 24
 tfs 32
 Timer::onTick 188
 Transaction::init 50
 Transaction::Transaction 49, 50, 51
 Uncopyable::operator= 39
 Uncopyable::Uncopyable 39
 unlock 66
 validateStudent 87
 Widget::onTick 189
 Widget::operator new 244
 Widget::operator+= 53
 Widget::operator= 53, 54, 55, 56, 107
 Widget::set_new_handler 243
 Widget::swap 108
 Window::blink 122
 Window::onResize 119
 workWithIterator 206, 207
 Year::Year 79
 exception specifications 85
Exceptional C++ xvii
Exceptional C++ Style xvii, xviii
 exceptions 113
 delete and 62

destructors and 44–48
 member swap and 112
 standard hierarchy for 264
 swallowing 46
 unused objects and 114
 exception-safe code 127–134
 copy-and-swap and 132
 legacy code and 133
 pimpl idiom and 131
 side effects and 132
 exception-safety guarantees 128–129
 explicit calls to base class functions 211
 explicit constructors 5, 85, 104
 generalized copy construction and 219
 explicit inline request 135
 explicit specification, of class names 162
 explicit type conversions vs. implicit 70–72
 expression templates 237
 expressions, implicit interfaces and 201

F

factoring code, out of templates 212–217
 factory function 40, 62, 69, 81, 146, 195
 Fallenstedt, Martin xix
 federation, of languages, C++ as 11–13
 Fehér, Attila F. xix
 final classes, in Java 43
 final methods, in Java 190
 fixed-size static buffers, problems of 196
 forms of new and delete 73–75
 FORTRAN 42
 forward iterators 227
 forward_iterator_tag 228
 forwarding functions 144, 160
 French, Donald xx
 friend functions 38, 85, 105, 135, 173, 223–225
 vs. member functions 98–102
 friendship
 in real life 105
 without needing special access rights 225
 Fruchterman, Thomas xix
 FUDGE_FACTOR 15
 Fuller, John xx
 function declarations, const in 18
 function objects
 definition of 6
 higher-order programming utilities
 and, in Boost 271
 functions
 convenience 100
 copying 57

defining 4
 deliberately not defining 38
 factory, see [factory function](#)
 forwarding 144, 160
 implicitly generated 34–37, 221
 disallowing 37–39
 inline, declaring 135
 member
 templated 218–222
 vs. non-member 104–105
 non-member
 templates and 222–226
 type conversions and 102–105, 222–226
 non-member non-friend, vs. member 98–102
 non-virtual, meaning 168
 return values, modifying 21
 signatures, explicit interfaces and 201
 static
 ctors and dtors and 52
 virtual, see [virtual functions](#)
 function-style casts 116

G

Gamma, Erich [xvii](#)
 Geller, Alan [xix](#)
 generalized assignment 220
 generalized copy constructors 219
 generative programming 237
 generic programming support, in Boost 271
 get, smart pointers and 70
 goddess, see [Urbano, Nancy L.](#)
 goto, delete and 62
 Graph library, in Boost 270
 grep, casts and 117
 guarantees, exception safety 128–129
 Gutnik, Gene [xix](#)

H

Handle classes 144–145
 handles 125
 dangling 126
 encapsulation and 125
 operator[] and 126
 returning 123–126
 has-a relationship 184
 hash tables, in TR1 266
 Hastings, Battle of 150
 Haugland, Solveig [xx](#)
 head scratching, avoiding 95
 header files, see [headers](#)

headers
 for declarations vs. for definitions 144
 inline functions and 135
 namespaces and 100
 of C++ standard library 101
 templates and 136
 usage, in this book 3
 hello world, template metaprogramming and 235
 Helm, Richard [xvii](#)
 Henney, Kevlin [xix](#)
 Hicks, Cory [xix](#)
 hiding names, see [name hiding](#)
 higher-order programming and function object utilities, in Boost 271
 highlighting, in this book 5

I

identity test 55
 if...else for types 230
 #ifdef 17
 #ifndef 17
 implementation-dependent behavior, warnings and 263
 implementations
 decoupling from interfaces 165
 default, danger of 163–167
 inheritance of 161–169
 of derived class constructors and destructors 137
 of Interface classes 147
 references 89
 std::max 135
 std::swap 106
 implicit inline request 135
 implicit interfaces 199–203
 implicit type conversions vs. explicit 70–72
 implicitly generated functions 34–37, 221
 disallowing 37–39
 #include directives 17
 compilation dependencies and 140
 incompatibilities, with built-in types 80
 incorrect code and efficiency 90
 infinite loop, in operator new 253
 inheritance
 accidental 165–166
 combining with templates 243–245
 common features and 164
 intuition and 151–155
 mathematics and 155
 mixin-style 244
 name hiding and 156–161
 of implementation 161–169
 of interface 161–169

- of interface vs. implementation 161–169
 - operator new and 253–254
 - penguins and birds and 151–153
 - private 187–192
 - protected 151
 - public 150–155
 - rectangles and squares and 153–155
 - redefining non-virtual functions
 - and 178–180
 - scopes and 156
 - sharing features and 164
 - inheritance, multiple 192–198
 - ambiguity and 192
 - combining public and private 197
 - deadly diamond 193
 - inheritance, private 214
 - combining with public 197
 - eliminating 189
 - for redefining virtual functions 197
 - meaning 187
 - vs. composition 188
 - inheritance, public
 - combining with private 197
 - is-a relationship and 150–155
 - meaning of 150
 - name hiding and 159
 - virtual inheritance and 194
 - inheritance, virtual 194
 - init function 60
 - initialization 4, 26–27
 - assignment vs. 6
 - built-in types 26–27
 - const members 29
 - const static members 14
 - default, unintended 59
 - in-class, of static const integral
 - members 14
 - local static objects 31
 - non-local static objects 30
 - objects 26–33
 - reference members 29
 - static members 242
 - virtual base classes and 194
 - vs. assignment 27–29, 114
 - with vs. without arguments 114
 - initialization order
 - class members 29
 - importance of 31
 - non-local statics 29–33
 - inline functions
 - see also [inlining](#)
 - address of 136
 - as request to compiler 135
 - debuggers and 139
 - declaring 135
 - headers and 135
 - optimizing compilers and 134
 - recursion and 136
 - vs. #define 16–17
 - vs. macros, efficiency and 16
 - inlining 134–139
 - constructors/destructors and 137–138
 - dynamic linking and 139
 - Handle classes and 148
 - inheritance and 137–138
 - Interface classes and 148
 - library design and 138
 - recompiling and 139
 - relinking and 139
 - suggested strategy for 139
 - templates and 136
 - time of 135
 - virtual functions and 136
 - input iterators 227
 - input_iterator_tag 228
 - input_iterator_tag<Iter*> 230
 - insomnia 150
 - instructions, reordering by compilers 76
 - integral types 14
 - Interface classes 145–147
 - interfaces
 - decoupling from implementations 165
 - definition of 7
 - design considerations 78–86
 - explicit, signatures and 201
 - implicit 199–203
 - expressions and 201
 - inheritance of 161–169
 - new types and 79–80
 - separating from implementations 140
 - template parameters and 199–203
 - undeclared 85
 - inter-language support, in Boost 272
 - internationalization, library support
 - for 264
 - invalid array index, undefined behavior
 - and 7
 - invariants
 - NVI and 171
 - over specialization 168
 - <iosfwd> 144
 - is-a relationship 150–155
 - is-implemented-in-terms-of 184–186, 187
 - istream_iterators 227
 - iterator categories 227–228
 - iterator_category 229
 - iterators as handles 125
 - iterators, vs. const_iterators 18
- ## J
- Jagdhari, Emily [xix](#)
 - Janert, Philipp [xix](#)
 - Java 7, 43, 76, 81, 100, 116, 118, 142, 145, 190, 194

Johnson, Ralph [xvii](#)
 Johnson, Tim [xviii, xix](#)
 Josuttis, Nicolai M. [xviii](#)

K

Kaelbling, Mike [xviii](#)
 Kakulapati, Gunavardhan [xix](#)
 Kalenkovich, Eugene [xix](#)
 Kennedy, Glenn [xix](#)
 Kernighan, Brian [xviii, xix](#)
 Kimura, Junichi [xviii](#)
 Kirman, Jak [xviii](#)
 Kirmse, Andrew [xix](#)
 Knox, Timothy [xviii, xix](#)
 Koenig lookup [110](#)
 Kourounis, Drosos [xix](#)
 Kreuzer, Gerhard [xix](#)

L

Laeuchli, Jesse [xix](#)
 Lambda library, in Boost [271](#)
 Langer, Angelika [xix](#)
 languages, other, compatibility with [42](#)
 Lanzetta, Michael [xix](#)
 late binding [180](#)
 layering, see [composition](#)
 layouts, objects vs. arrays [73](#)
 Lea, Doug [xviii](#)
 leaks, exception-safe code and [127](#)
 Leary-Coutu, Chanda [xx](#)
 Lee, Sam [xix](#)
 legacy code, exception-safety and [133](#)
 Lejter, Moises [xviii, xx](#)
 lemur, ring-tailed [196](#)
 Lewandowski, Scott [xviii](#)
 lhs, as parameter name [8](#)
 Li, Greg [xix](#)
 link-time errors [39, 44](#)
 link-time inlining [135](#)
 list [186](#)
 local static objects
 definition of [30](#)
 initialization of [31](#)
 locales [264](#)
 locks, RAI and [66–68](#)
 logic_error class [113](#)
 logically const member functions [22–23](#)

M

mailing list for Scott Meyers [xvi](#)

maintenance
 common base classes and [164](#)
 delete and [62](#)
 managing resources, see [resource management](#)
 Manis, Vincent [xix](#)
 Marin, Alex [xix](#)
 math and numerics utilities, in Boost [271](#)
 mathematical functions, in TR1 [267](#)
 mathematics, inheritance and [155](#)
 matrix operations, optimizing [237](#)
 Matthews, Leon [xix](#)
 max, std, implementation of [135](#)
 Meadowbrooke, Chrysta [xix](#)
 meaning
 of classes without virtual functions [41](#)
 of composition [184](#)
 of non-virtual functions [168](#)
 of pass-by-value [6](#)
 of private inheritance [187](#)
 of public inheritance [150](#)
 of pure virtual functions [162](#)
 of references [91](#)
 of simple virtual functions [163](#)
 measuring encapsulation [99](#)
 Meehan, Jim [xix](#)
 member data, see [data members](#)
 member function templates [218–222](#)
 member functions
 bitwise const [21–22](#)
 common design errors [168–169](#)
 const [19–25](#)
 duplication and [23–25](#)
 encapsulation and [99](#)
 implicitly generated [34–37, 221](#)
 disallowing [37–39](#)
 logically const [22–23](#)
 private [38](#)
 protected [166](#)
 vs. non-member functions [104–105](#)
 vs. non-member non-friends [98–102](#)
 member initialization
 for const static integral members [14](#)
 lists [28–29](#)
 vs. assignment [28–29](#)
 order [29](#)
 memory allocation
 arrays and [254–255](#)
 error handling for [240–246](#)
 memory leaks, new expressions and [256](#)
 memory management
 functions, replacing [247–252](#)
 multithreading and [239, 253](#)
 utilities, in Boost [272](#)
 metaprogramming, see [template metaprogramming](#)

Meyers, Scott
 mailing list for [xvi](#)
 web site for [xvi](#)
 mf, as identifier [9](#)
 Michaels, Laura [xviii](#)
 Mickelsen, Denise [xx](#)
 minimizing compilation
 dependencies [140–148, 190](#)
 Mittal, Nishant [xix](#)
 mixed-mode arithmetic [103, 104, 222–226](#)
 mixin-style inheritance [244](#)
 modeling is-implemented-in-terms-
 of [184–186](#)
 modifying function return values [21](#)
 Monty Python, allusion to [91](#)
 Moore, Vanessa [xx](#)
More Effective C++ [273, 273–274](#)
 compared to *Effective C++* [273](#)
 contents of [273–274](#)
More Exceptional C++ [xvii](#)
 Moroff, Hal [xix](#)
 MPL library, in Boost [270, 271](#)
 multiparadigm programming language,
 C++ as [11](#)
 multiple inheritance, see [inheritance](#)
 multithreading
 memory management routines
 and [239, 253](#)
 non-const static objects and [32](#)
 treatment in this book [9](#)
 mutable [22–23](#)
 mutexes, RAI and [66–68](#)

N

Nagler, Eric [xix](#)
 Nahil, Julie [xx](#)
 name hiding
 inheritance and [156–161](#)
 operators new/delete and [259–261](#)
 using declarations and [159](#)
 name lookup
 this-> and [210](#)
 using declarations and [211](#)
 name shadowing, see [name hiding](#)
 names
 accessing in templated bases [207–212](#)
 available in both C and C++ [3](#)
 dependent [204](#)
 hidden by derived classes [263](#)
 nested, dependent [204](#)
 non-dependent [204](#)
 namespaces [110](#)
 headers and [100](#)
 namespace pollution in a class [166](#)
 Nancy, see [Urbano, Nancy L.](#)

Nauroth, Chris [xix](#)
 nested dependent names [204](#)
 nested dependent type names, typename
 and [205](#)
 new
 see also [operator new](#)
 expressions, memory leaks and [256](#)
 forms of [73–75](#)
 operator new and [73](#)
 relationship to constructors [73](#)
 smart pointers and [75–77](#)
 new types, interface design and [79–80](#)
 new-handler [240–247](#)
 definition of [240](#)
 deinstalling [241](#)
 identifying [253](#)
 new-handling functions, behavior of [241](#)
 new-style casts [117](#)
 noncopyable base class, in Boost [39](#)
 non-dependent names [204](#)
 non-local static objects, initialization
 of [30](#)
 non-member functions
 member functions vs. [104–105](#)
 templates and [222–226](#)
 type conversions and [102–105, 222–226](#)
 non-member non-friend functions [98–102](#)
 non-type parameters [213](#)
 non-virtual
 functions [178–180](#)
 static binding of [178](#)
 interface idiom, see [NVI](#)
 nothrow guarantee, the [129](#)
 nothrow new [246](#)
 null pointer
 deleting [255](#)
 dereferencing [6](#)
 set_new_handler and [241](#)
 NVI [170–171, 183](#)

O

object-oriented C++, as sublanguage of
 C++ [12](#)
 object-oriented principles, encapsulation
 and [99](#)
 objects
 alignment of [249–250](#)
 clustering [251](#)
 compilation dependencies and [143](#)
 copying all parts [57–60](#)
 defining [4](#)
 definitions, postponing [113–116](#)
 handles to internals of [123–126](#)
 initialization, with vs. without
 arguments [114](#)
 layout vs. array layout [73](#)

- multiple addresses for 118
 - partial copies of 58
 - placing in shared memory 251
 - resource management and 61–66
 - returning, vs. references 90–94
 - size, pass-by-value and 89
 - sizes, determining 141
 - vs. variables 3
 - Oldham, Jeffrey D. [xix](#)
 - old-style casts 117
 - operations, reordering by compilers 76
 - operator delete 84
 - see also [delete](#)
 - behavior of 255
 - efficiency of 248
 - name hiding and 259–261
 - non-member, pseudocode for 255
 - placement 256–261
 - replacing 247–252
 - standard forms of 260
 - virtual destructors and 255
 - operator delete[] 84, 255
 - operator new 84
 - see also [new](#)
 - arrays and 254–255
 - bad_alloc and 246, 252
 - behavior of 252–255
 - efficiency of 248
 - infinite loop within 253
 - inheritance and 253–254
 - member, and “wrongly sized” requests 254
 - name hiding and 259–261
 - new-handling functions and 241
 - non-member, pseudocode for 252
 - out-of-memory conditions and 240–241, 252–253
 - placement 256–261
 - replacing 247–252
 - returning 0 and 246
 - standard forms of 260
 - std::bad_alloc and 246, 252
 - operator new[] 84, 254–255
 - operator() (function call operator) 6
 - operator=
 - const members and 36–37
 - default implementation 35
 - implicit generation 34
 - reference members and 36–37
 - return value of 52–53
 - self-assignment and 53–57
 - when not implicitly generated 36–37
 - operator[] 126
 - overloading on const 19–20
 - return type of 21
 - optimization
 - by compilers 94
 - during compilation 134
 - inline functions and 134
 - order
 - initialization of non-local statics 29–33
 - member initialization 29
 - ostream_iterators 227
 - other languages, compatibility with 42
 - output iterators 227
 - output_iterator_tag 228
 - overloading
 - as if...else for types 230
 - on const 19–20
 - std::swap 109
 - overrides of virtuals, preventing 189
 - ownership transfer 68
- ## P
- Pal, Balog [xix](#)
 - parameters
 - see also [pass-by-value](#), [pass-by-reference](#)
 - default 180–183
 - evaluation order 76
 - non-type, for templates 213
 - type conversions and, see [type conversions](#)
 - Pareto Principle, see [80-20 rule](#)
 - parsing problems, nested dependent names and 204
 - partial copies 58
 - partial specialization
 - function templates 109
 - std::swap 108
 - parts, of objects, copying all 57–60
 - pass-by-reference, efficiency and 87
 - pass-by-reference-to-const, vs pass-by-value 86–90
 - pass-by-value
 - copy constructor and 6
 - efficiency of 86–87
 - meaning of 6
 - object size and 89
 - vs. pass-by-reference-to-const 86–90
 - patterns
 - see [design patterns](#)
 - Pedersen, Roger E. [xix](#)
 - penguins and birds 151–153
 - performance, see [efficiency](#)
 - Persephone [ix](#), [xx](#), 36
 - pessimization 93
 - physical constness, see [const](#), [bitwise](#)
 - pimpl idiom
 - definition of 106
 - exception-safe code and 131

placement delete, see [operator delete](#)
 placement new, see [operator new](#)
 Plato 87
 pointer arithmetic and undefined behavior 119
 pointers
 see also [smart pointers](#)
 as handles 125
 bitwise const member functions and 21
 compilation dependencies and 143
 const 17
 in headers 14
 null, dereferencing 6
 template parameters and 217
 to single vs. multiple objects, and delete 73
 polymorphic base classes, destructors and 40–44
 polymorphism 199–201
 compile-time 201
 runtime 200
 Pool library, in Boost 250, 251
 postponing variable definitions 113–116
 Prasertsith, Chuti xx
 preconditions, NVI and 171
 pregnancy, exception-safe code and 133
 private data members, why 94–98
 private inheritance, see [inheritance](#)
 private member functions 38
 private virtual functions 171
 properties 97
 protected
 data members 97
 inheritance, see [inheritance](#)
 member functions 166
 members, encapsulation of 97
 public inheritance, see [inheritance](#)
 pun, really bad 152
 pure virtual destructors
 defining 43
 implementing 43
 pure virtual functions 43
 defining 162, 166–167
 meaning 162

R

Rabbani, Danny xix
 Rabinowitz, Marty xx
 RAI 63, 70, 243
 classes 72
 copying behavior and 66–69
 encapsulation and 72
 mutexes and 66–68
 random access iterators 227

random number generation, in TR1 267
 random_access_iterator_tag 228
 RCSP, see [smart pointers](#)
 reading uninitialized values 26
 rectangles and squares 153–155
 recursive functions, inlining and 136
 redefining inherited non-virtual functions 178–180
 Reed, Kathy xx
 Reeves, Jack xix
 references
 as handles 125
 compilation dependencies and 143
 functions returning 31
 implementation 89
 meaning 91
 members, initialization of 29
 returning 90–94
 to static object, as function return value 92–94
 register usage, objects and 89
 regular expressions, in TR1 266
 reinterpret_cast 117, 249
 see also [casting](#)
 relationships
 has-a 184
 is-a 150–155
 is-implemented-in-terms-of 184–186, 187
 reordering operations, by compilers 76
 replacing definitions with declarations 143
 replacing new/delete 247–252
 replication, see [duplication](#)
 reporting, bugs in this book xvi
 Resource Acquisition Is Initialization, see [RAII](#)
 resource leaks, exception-safe code and 127
 resource management
 see also [RAII](#)
 copying behavior and 66–69
 objects and 61–66
 raw resource access and 69–73
 resources, managing objects and 69–73
 return by reference 90–94
 return types
 const 18
 objects vs. references 90–94
 of operator[] 21
 return value of operator= 52–53
 returning handles 123–126
 reuse, see [code reuse](#)
 revenge, compilers taking 58
 rhs, as parameter name 8

Roze, Mike [xix](#)
 rule of 80-20 [139, 168](#)
 runtime
 errors [152](#)
 inlining [135](#)
 polymorphism [200](#)

S

Saks, Dan [xviii](#)
 Santos, Eugene, Jr. [xviii](#)
 Satch [36](#)
Satyricon [vii](#)
 Schirpelz, Jeff [xix](#)
 Schirripa, Steve [xix](#)
 Schober, Hendrik [xviii, xix](#)
 Schroeder, Sandra [xx](#)
 scoped_array [65, 216, 272](#)
 scopes, inheritance and [156](#)
 sealed classes, in C# [43](#)
 sealed methods, in C# [190](#)
 second edition, see [2nd edition](#)
 self-assignment, operator= and [53-57](#)
 set [185](#)
 set_new_handler
 class-specific, implementing [243-245](#)
 using [240-246](#)
 set_unexpected function [129](#)
 shadowing, names, see [name shadowing](#)
 Shakespeare, William [156](#)
 shared memory, placing objects in [251](#)
 shared_array [65](#)
 shared_ptr implementation in Boost,
 costs [83](#)
 sharing code, see [duplication, avoiding](#)
 sharing common features [164](#)
 Shewchuk, John [xviii](#)
 side effects, exception safety and [132](#)
 signatures
 definition of [3](#)
 explicit interfaces and [201](#)
 simple virtual functions, meaning of [163](#)
 Singh, Siddhartha [xix](#)
 Singleton pattern [31](#)
 size_t [3](#)
 sizeof [253, 254](#)
 empty classes and [190](#)
 freestanding classes and [254](#)
 sizes
 of freestanding classes [254](#)
 of objects [141](#)
 sleeping pills [150](#)
 slist [227](#)
 Smallberg, David [xviii, xix](#)

Smalltalk [142](#)
 smart pointers [63, 64, 70, 81, 121, 146, 237](#)
 see also [std::auto_ptr](#) and [tr1::shared_ptr](#)
 get and [70](#)
 in Boost [65, 272](#)
 web page for [xvii](#)
 in TR1 [265](#)
 newed objects and [75-77](#)
 type conversions and [218-220](#)

Socrates [87](#)
Some Must Watch While Some Must Sleep [150](#)

Somers, Jeff [xix](#)
 specialization
 invariants over [168](#)
 partial, of [std::swap](#) [108](#)
 total, of [std::swap](#) [107, 108](#)
 specification, see [interfaces](#)
 squares and rectangles [153-155](#)
 standard exception hierarchy [264](#)
 standard forms of operator new/delete [260](#)
 standard library, see [C++ standard library](#), [C standard library](#)
 standard template library, see [STL](#)
 Stasko, John [xviii](#)
 statements using new, smart pointers
 and [75-77](#)

static
 binding
 of default parameters [182](#)
 of non-virtual functions [178](#)
 objects, returning references to [92-94](#)
 type, definition of [180](#)
 static functions, ctors and dtors and [52](#)
 static members
 const member functions and [21](#)
 definition [242](#)
 initialization [242](#)

static objects
 definition of [30](#)
 multithreading and [32](#)

static_cast [25, 82, 117, 119, 249](#)
 see also [casting](#)

std namespace, specializing templates
 in [107](#)

std::auto_ptr [63-65, 70](#)
 conversion to [tr1::shared_ptr](#) and [220](#)
 delete [] and [65](#)
 pass by const and [220](#)

std::auto_ptr, deleter support and [68](#)
 std::char_traits [232](#)
 std::iterator_traits, pointers and [230](#)
 std::list [186](#)
 std::max, implementation of [135](#)
 std::numeric_limits [232](#)

std::set 185
 std::size_t 3
 std::swap
 see also [swap](#)
 implementation of 106
 overloading 109
 partial specialization of 108
 total specialization of 107, 108
 std::tr1, see [TR1](#)
 stepping through functions, inlining
 and 139
 STL
 allocators 240
 as sublanguage of C++ 12
 containers, swap and 108
 definition of 6
 iterator categories in 227–228
 Strategy pattern 171–177
 string and text utilities, in Boost 271
 strong guarantee, the 128
 Stroustrup, Bjarne [xvii](#), [xviii](#)
 Stroustrup, Nicholas [xix](#)
 Sutter, Herb [xvii](#), [xviii](#), [xix](#)
 swallowing exceptions 46
 swap 106–112
 see also [std::swap](#)
 calling 110
 exceptions and 112
 STL containers and 108
 when to write 111
 symbols, available in both C and C++ 3

T

template C++, as sublanguage of C++ 12
 template metaprogramming 233–238
 efficiency and 233
 hello world in 235
 pattern implementations and 237
 support in Boost 271
 support in TR1 267
 Template Method pattern 170
 templates
 code bloat, avoiding in 212–217
 combining with inheritance 243–245
 defining 4
 errors, when detected 212
 expression 237
 headers and 136
 in std, specializing 107
 inlining and 136
 instantiation of 222
 member functions 218–222
 names in base classes and 207–212
 non-type parameters 213
 parameters, omitting 224
 pointer type parameters and 217
 shorthand for 224
 specializations 229, 235
 partial 109, 230
 total 107, 209
 type conversions and 222–226
 type deduction for 223
 temporary objects, eliminated by
 compilers 94
 terminology, used in this book 3–8
 testing and correctness, Boost support
 for 272
 text and string utilities, in Boost 271
 third edition, see [3rd edition](#)
 this->, to force base class lookup 210
 threading, see [multithreading](#)
 Tilly, Barbara [xviii](#)
 TMP, see [template metaprogramming](#)
 Tondo, Clovis [xviii](#)
 Topic, Michael [xix](#)
 total class template specialization 209
 total specialization of std::swap 107, 108
 total template specializations 107
 TR1 9, 264–267
 array component 267
 bind component 266
 Boost and 9–10, 268, 269
 boost as synonym for std::tr1 268
 C99 compatibility component 267
 function component 265
 hash tables component 266
 math functions component 267
 mem_fn component 267
 random numbers component 267
 reference_wrapper component 267
 regular expression component 266
 result_of component 267
 smart pointers component 265
 support for TMP 267
 tuples component 266
 type traits component 267
 URL for information on 268
 tr1::array 267
 tr1::bind 175, 266
 tr1::function 173–175, 265
 tr1::mem_fn 267
 tr1::reference_wrapper 267
 tr1::result_of 267
 tr1::shared_ptr 53, 64–65, 70, 75–77
 construction from other smart pointers
 and 220
 cross-DLL problem and 82
 delete [] and 65
 deleter support in 68, 81–83
 member template ctors in 220–221
 tr1::tuple 266

tr1::unordered_map 43, 266
 tr1::unordered_multimap 266
 tr1::unordered_multiset 266
 tr1::unordered_set 266
 tr1::weak_ptr 265
 traits classes 226–232
 transfer, ownership 68
 translation unit, definition of 30
 Trux, Antoine xviii
 Tsao, Mike xix
 tuples, in TR1 266
 type conversions 85, 104
 explicit ctors and 5
 implicit 104
 implicit vs. explicit 70–72
 non-member functions and 102–105, 222–226
 private inheritance and 187
 smart pointers and 218–220
 templates and 222–226
 type deduction, for templates 223
 type design 78–86
 type traits, in TR1 267
 typedef, typename and 206–207
 typedefs, new/delete and 75
 typeid 50, 230, 234, 235
 typelists 271
 typename 203–207
 compiler variations and 207
 typedef and 206–207
 vs. class 203
 types
 built-in, initialization 26–27
 compatible, accepting all 218–222
 if...else for 230
 integral, definition of 14
 traits classes and 226–232

U

undeclared interface 85
 undefined behavior
 advance and 231
 array deletion and 73
 casting + pointer arithmetic and 119
 definition of 6
 destroyed objects and 91
 exceptions and 45
 initialization order and 30
 invalid array index and 7
 multiple deletes and 63, 247
 null pointers and 6
 object deletion and 41, 43, 74
 uninitialized values and 26
 undefined values of members before construction and after destruction 50

unexpected function 129
 uninitialized
 data members, virtual functions and 49
 values, reading 26
 unnecessary objects, avoiding 115
 unused objects
 cost of 113
 exceptions and 114
 Urbano, Nancy L. vii, xviii, xx
 see also goddess
 URLs
 Boost 10, 269, 272
 Boost smart pointers xvii
 Effective C++ errata list xvi
 Effective C++ TR1 Info. Page 268
 Greg Comeau's C/C++ FAQ xviii
 Scott Meyers' mailing list xvi
 Scott Meyers' web site xvi
 this book's errata list xvi
 usage statistics, memory management and 248
 using declarations
 name hiding and 159
 name lookup and 211

V

valarray 264
 value, pass by, see *pass-by-value*
 Van Wyk, Chris xviii, xix
 Vandevoorde, David xviii
 variable, vs. object 3
 variables definitions, postponing 113–116
 vector template 75
 Viciano, Paco xix
 virtual base classes 193
 virtual constructors 146, 147
 virtual destructors
 operator delete and 255
 polymorphic base classes and 40–44
 virtual functions
 alternatives to 169–177
 ctors/dtors and 48–52
 default implementations and 163–167
 default parameters and 180–183
 dynamic binding of 179
 efficiency and 168
 explicit base class qualification and 211
 implementation 42
 inlining and 136
 language interoperability and 42
 meaning of none in class 41
 preventing overrides 189
 private 171
 pure, see *pure virtual functions*
 simple, meaning of 163

- uninitialized data members and 49
- virtual inheritance, see [inheritance](#)
- virtual table 42
- virtual table pointer 42
- Vlissides, John [xvii](#)
- vptr 42
- vtbl 42

W

- Wait, John [xx](#)
- warnings, from compiler 262–263
 - calls to virtuals and 50
 - inlining and 136
 - partial copies and 58
- web sites, see [URLs](#)
- Widget class, as used in this book 8
- Wiegers, Karl [xix](#)
- Wilson, Matthew [xix](#)
- Wizard of Oz*, allusion to 154

X

- XP, allusion to 225
- XYZ Airlines 163

Z

- Zabluda, Oleg [xviii](#)
- Zolman, Leor [xviii](#), [xix](#)