More effective c

Table of contents : Cover......Page 1Contents......Page 10Acknowledgments......Page 12Introduction......Page 18 Item 1: Distinguish between pointers and references......Page 26 Item 2: Prefer C++-style casts......Page 29 Item 3: Never treat arrays polymorphically......Page 33 Item 4: Avoid gratuitous default constructors......Page 36 Item 5: Be wary of user-defined conversion functions......Page 41 Item 6: Distinguish between prefix and postfix forms of increment and decrement operators......Page 48 Item 7: Never overload &&, ||, or ,......Page 52 Item 8: Understand the different meanings of new and delete......Page 55Exceptions......Page 61 Item 9: Use destructors to prevent resource leaks......Page 62 Item 10: Prevent resource leaks in constructors......Page 67 Item 11: Prevent exceptions from leaving destructors......Page 75 Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function......Page 78 Item 13: Catch exceptions by reference......Page 85 Item 14: Use exception specifications judiciously......Page 89 Item 15: Understand the costs of exception handling......Page 95Efficiency......Page 98 Item 16: Remember the 80-20 rule......Page 99 Item 17: Consider using lazy evaluation......Page 102 Item 18: Amortize the cost of expected computations......Page 110 Item 19: Understand the origin of temporary objects......Page 115 Item 20: Facilitate the return value optimization......Page 118 Item 21: Overload to avoid implicit type conversions......Page 122 Item 22: Consider using op= instead of stand-alone op......Page 124 Item 23: Consider alternative libraries......Page 127 Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI......Page 130 Item 25: Virtualizing constructors and non-member functions......Page 140 Item 26: Limiting the number of objects of a class......Page 147 Item 27: Requiring or prohibiting heap-based objects......Page 162 Item 28: Smart pointers......Page 176 Item 29: Reference counting......Page 200 Item 30: Proxy classes......Page 230 Item 31: Making functions virtual with respect to more than one object......Page 245 Item 32: Program in the future tense......Page 269 Item 33: Make non-leaf classes abstract......Page 275 Item 34: Understand how to combine C++ and C in the same program......Page 287 Item 35: Familiarize yourself with the language standard......Page 294Recommended Reading......Page 302An auto_ptr Implementation......Page 308 A......Page 312 C......Page 313 D......Page 315 E......Page 317 F......Page 318 I......Page 319 M......Page 320 O......Page 322 P......Page 323 R......Page 325 S......Page 326 T......Page 327 V......Page 328 Y......Page 329 R......Page 330 C......Page 331 M......Page 332 R......Page 333 U......Page 334 W......Page 335 Citation preview Praise for More Effective C++: 35 New Ways to Improve Your Programs and Designs "This is an enlightening book on many aspects of C++: both the regions of the language you seldom visit, and the familiar ones you THOUGHT you understood. Only by understanding deeply how the C++ compiler interprets your code can you hope to write robust software using this language. This book is an invaluable resource for gaining that level of understanding. After reading this book, I feel like I've been through a code review with a master C++ programmer, and picked up many of his most valuable insights." — Fred Wild, Vice President of Technology, Advantage Software Technologies "This book includes a great collection of important techniques for writing programs that use C++ well. It explains how to design and implement the ideas, and what hidden pitfalls lurk in some obvious alternative designs. It also includes clear explanations of features recently added to C++. Anyone who wants to use these new features will want a copy of this book close at hand for ready reference." — Christopher J. Van Wyk, Professor, Mathematics and Computer Science, Drew University "Industrial strength C++ at its best. The perfect companion to those who have read Effective C++." — Eric Nagler, C++ Instructor and Author, University of California Santa Cruz Extension "More Effective C++ is a thorough and valuable follow-up to Scott's first book, Effective C++. I believe that every professional C++ developer should read and commit to memory the tips in both Effective C++ and More Effective C++. I've found that the tips cover poorly understood, yet important and sometimes arcane facets of the language. I strongly recommend this book, along with his first, to developers, testers, and managers ... everyone can benefit from his expert knowledge and excellent presentation." — Steve Burkett, Software Consultant This page intentionally left blank More Effective C++ 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 Haggar, 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 Visit www.awprofessional.com/series/professionalcomputing for more information about these titles. More Effective C++ 35 New Ways to Improve Your Programs and Designs Scott Meyers Upper Saddle River, NJ • Boston • Indianapolis • San Francisco New York • Toronto • Montreal • London • Munich • Paris • Madrid Cape Town • Sydney • Tokyo • Singapore • Mexico City Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals. The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein. The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact: U.S. Corporate and Government Sales (800) 382-3419 For sales outside the United States please contact: International Sales Library of Congress Cataloging-in-Publication Data Meyers, Scott (Scott Douglas) More effective C++ : 35 new ways to improve your programs and designs / Scott Meyers. p. cm. — (Addison-Wesley professional computing series) Includes bibliographical references and index. ISBN 0-201-63371-X (paperback: alk. paper) 1. C++ (Computer program language) I. Series. QA76.73.C153M495 1996 005.13'3—dc20 95-47354 CIP Copyright © 1996 by Addison-Wesley All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to: Pearson Education, Inc. Rights and Contracts Department 501 Boylston Street, Suite 900 Boston, MA 02116 Fax (617) 671-3447 ISBN-13: 978-0-201-63371-9 ISBN-10: 0-201-63371-X Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts. Twenty-fourth printing April 2007 For Clancy, my favorite enemy within. This page intentionally left blank Contents Acknowledgments Introduction 1 Basics 9 Item 1: Distinguish between pointers and references. 9 Item 2: Prefer C++-style casts. 12 Item 3: Never treat arrays polymorphically. 16 Item 4: Avoid gratuitous default constructors. 19 Operators 9 24 Item 5: Be wary of user-defined conversion functions. Item 6: Distinguish between prefix and postfix forms of increment and decrement operators. 31 Item 7: Never overload &&, ||, or ,. 35 Item 8: Understand the different meanings of new and delete. 38 Exceptions Item 9: Use destructors to prevent resource leaks. 24 44 45 Item 10: Prevent resource leaks in constructors. 50 Item 11: Prevent exceptions from leaving destructors. 58 Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function. 61 Item 13: Catch exceptions by reference. 68 Item 14: Use exception specifications judiciously. 72 Item 15: Understand the costs of exception handling. 78 x Contents Efficiency 81 Item 16: Remember the 80-20 rule. 82 Item 17: Consider using lazy evaluation. 85 Item 18: Amortize the cost of expected computations. 93 Item 19: Understand the origin of temporary objects. 98 Item 20: Facilitate the return value optimization. 101 Item 21: Overload to avoid implicit type conversions. 105 Item 22: Consider using op= instead of stand-alone op. 107 Item 23: Consider alternative libraries. 110 Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI. 113 Techniques 123 Item 25: Virtualizing constructors and non-member functions. 123 Item 26: Limiting the number of objects of a class. 130 Item 27: Requiring or prohibiting heap-based objects. 145 Item 28: Smart pointers. 159 Item 29: Reference counting. 183 Item 30: Proxy classes. 213 Item 31: Making functions virtual with respect to more than one object. 228 Miscellany 252 Item 32: Program in the future tense. 252 Item 33: Make non-leaf classes abstract. 258 Item 34: Understand how to combine C++ and C in the same program. 270 Item 35: Familiarize yourself with the language standard. 277 Recommended Reading 285 An auto_ptr Implementation 291 General Index 295 Index of Example Classes, Functions, and Templates 313 Acknowledgments A great number of people helped bring this book into existence. Some contributed ideas for technical topics, some helped with the process of producing the book, and some just made life more fun while I was working on it. When the number of contributors to a book is large, it is not uncommon to dispense with individual contributors in favor of a generic "Contributors to this book are too numerous to mention." I prefer to follow the expansive lead of John L. Hennessy and David A. Patterson in Computer Architecture: A Quantitative Approach (Morgan Kaufmann, first edition 1990). In addition to motivating the comprehensive acknowledgments that follow, their book provides hard data for the 90-10 rule, which I refer to in Item 16. The Items With the exception of direct quotations, all the words in this book are mine. However, many of the ideas I discuss came from others. I have done my best to keep track of who contributed what, but I know I have included information from sources I now fail to recall, foremost among them many posters to the Usenet newsgroups comp.lang.c++ and comp.std.c++. Many ideas in the C++ community have been developed independently by many people. In what follows, I note only where I was exposed to particular ideas, not necessarily where those ideas originated. Brian Kernighan suggested the use of macros to approximate the syntax of the new C++ casting operators I describe in Item 2. In Item 3, my warning about deleting an array of derived class objects through a base class pointer is based on material in Dan Saks' "Gotchas" talk, which he's given at several conferences and trade shows. xii Acknowledgments In Item 5, the proxy class technique for preventing unwanted application of single-argument constructors is based on material in Andrew Koenig's column in the January 1994 C++ Report. James Kanze made a posting to comp.lang.c++ on implementing postfix increment and decrement operators via the corresponding prefix functions; I use his technique in Item 6. David Cok, writing me about material I covered in Effective C++, brought to my attention the distinction between operator new and the new operator that is the crux of Item 8. Even after reading his letter, I didn't really understand the distinction, but without his initial prodding, I probably still wouldn't. The notion of using destructors to prevent resource leaks (used in Item 9) comes from section 15.3 of Margaret A. Ellis' and Bjarne Stroustrup's The Annotated C++ Reference Manual (see page 285). There the technique is called resource acquisition is initialization. Tom Cargill suggested I shift the focus of the approach from resource acquisition to resource release. Some of my discussion in Item 11 was inspired by material in Chapter 4 of Taligent's Guide to Designing Programs (Addison-Wesley, 1994). My description of over-eager memory allocation for the DynArray class in Item 18 is based on Tom Cargill's article, "A Dynamic vector is harder than it looks," in the June 1992 C++ Report. A more sophisticated design for a dynamic array class can be found in Cargill's follow-up column in the January 1994 C++ Report. Item 21 was inspired by Brian Kernighan's paper, "An AWK to C++ Translator," at the 1991 USENIX C++ Conference. His use of overloaded operators (sixty-seven of them!) to handle mixed-type arithmetic operations, though designed to solve a problem unrelated to the one I explore in Item 21, led me to consider multiple overloadings as a solution to the problem of temporary creation. In Item 26, my design of a template class for counting objects is based on a posting to comp.lang.c++ by Jamshid Afshar. The idea of a mixin class to keep track of pointers from operator new (see Item 27) is based on a suggestion by Don Box. Steve Clamage made the idea practical by explaining how dynamic_cast can be used to find the beginning of memory for an object. The discussion of smart pointers in Item 28 is based in part on Steven Buroff's and Rob Murray's C++ Oracle column in the October 1993 C++ Report; on Daniel R. Edelson's classic paper, "Smart Pointers: They're Smart, but They're Not Pointers," in the proceedings of the 1992 USENIX C++ Conference; on section 15.9.1 of Bjarne Stroustrup's The Design and Evolution of C++ (see page 285); on Gregory Colvin's "C++ Memory Management" class notes from C/C++ Solutions '95; and on Cay Horstmann's column in the March-April 1993 issue of the C++ Report. I developed some of the material myself, though. Really. In Item 29, the use of a base class to store reference counts and of smart pointers to manipulate those counts is based on Rob Murray's discussions of the same topics in sections 6.3.2 and 7.4.2, respectively, of his C++ Strategies and Tactics (see page 286). The design for adding reference counting to existing classes follows that presented by Cay Horstmann in his March-April 1993 column in the C++ Report. In Item 30, my discussion of lvalue contexts is based on comments in Dan Saks' column in the C User's Journal (now the C/C++ Users Journal) of January 1993. The observation that non-proxy member functions are unavailable when called through proxies comes from an unpublished paper by Cay Horstmann. The use of runtime type information to build vtbl-like arrays of function pointers (in Item 31) is based on ideas put forward by Bjarne Stroustrup in postings to comp.lang.c++ and in section 13.8.1 of his The Design and Evolution of C++ (see page 285). The material in Item 33 is based on several of my C++ Report columns in 1994 and 1995. Those columns, in turn, included comments I received from Klaus Kreft about how to use dynamic_cast to implement a virtual operator= that detects arguments of the wrong type. Much of the material in Item 34 was motivated by Steve Clamage's article, "Linking C++ with other languages," in the May 1992 C++ Report. In that same Item, my treatment of the problems caused by functions like strdup was motivated by an anonymous reviewer. The Book Reviewing draft copies of a book is hard — and vitally important — work. I am grateful that so many people were willing to invest their time and energy on my behalf. I am especially grateful to Jill Huchital, Tim Johnson, Brian Kernighan, Eric Nagler, and Chris Van Wyk, as they read the book (or large portions of it) more than once. In addition to these gluttons for punishment, complete drafts of the manuscript were read by Katrina Avery, Don Box, Steve Burkett, Tom Cargill, Tony Davis, Carolyn Duby, Bruce Eckel, Read Fleming, Cay Horstmann, James Kanze, Russ Paielli, Steve Rosenthal, Robin Rowe, Dan Saks, Chris Sells, Webb Stacy, Dave Swift, Steve Vinoski, and Fred Wild. Partial drafts were reviewed by Bob Beauchaine, Gerd Hoeren, Jeff Jackson, and Nancy L. Urbano. Each of these reviewers made comments that greatly improved the accuracy, utility, and presentation of the material you find here. Once the book came out, I received corrections and suggestions from many people: Luis Kida, John Potter, Tim Uttormark, Mike Fulkerson, Dan Saks, Wolfgang Glunz, Clovis Tondo, Michael Loftus, Liz Hanks, Wil Evers, Stefan Kuhlins, Jim McCracken, Alan Duchan, John Jacobsma, Ramesh Nagabushnam, Ed Willink, Kirk Swenson, Jack Reeves, Doug Schmidt, Tim Buchowski, Paul Chisholm, Andrew Klein, Eric Nagler, Jeffrey Smith, Sam Bent, Oleg Shteynbuk, Anton Doblmaier, Ulf Michaelis, Sekhar Muddana, Michael Baker, Yechiel Kimchi, David Papurt, Ian Haggard, Robert Schwartz, David Halpin, Graham Mark, David Barrett, Damian Kanarek, Ron Coutts, Lance Whitesel, Jon Lachelt, Cheryl Ferguson, Munir Mahmood, Klaus-Georg Adams, David Goh, Chris Morley, Rainer Baumschlager, Christopher Tavares, Brian Kernighan, Charles Green, Mark Rodgers, Bobby Schmidt, Sivaramakrishnan J., Eric Anderson, Phil Brabbin, Feliks Kluzniak, Evan McLean, Kurt Miller, Niels Dekker, Balog Pal, Dean Stanton, William Mattison, Chulsu Park, Pankaj Datta, John Newell, Ani Taggu, Christopher Creutzi, Chris Wineinger, Alexander Bogdanchikov, Michael Tegtmeyer, Aharon Robbins, Davide Gennaro, Adrian Spermezan, Matthias Hofmann, Chang Chen, John Wismar, Mark Symonds, Thomas Kim, Ita Ryan, and Rice Yeh. Their suggestions allowed me to improve More Effective C++ in updated printings (such as this one), and I greatly appreciate their help. During preparation of this book, I faced many questions about the emerging ISO/ANSI standard for C++, and I am grateful to Steve Clamage and Dan Saks for taking the time to respond to my incessant email queries. John Max Skaller and Steve Rumsby conspired to get me the HTML for the draft ANSI C++ standard before it was widely available. Vivian Neou pointed me to the Netscape WWW browser as a stand-alone HTML viewer under (16 bit) Microsoft Windows, and I am deeply grateful to the folks at Netscape Communications for making their fine viewer freely available on such a pathetic excuse for an operating system. Bryan Hobbs and Hachemi Zenad generously arranged to get me a copy of the internal engineering version of the MetaWare C++ compiler so I could check the code in this book using the latest features of the language. Cay Horstmann helped me get the compiler up and running in the very foreign world of DOS and DOS extenders. Borland (now Inprise) provided a beta copy of their most advanced compiler, and Eric Nagler and Chris Sells provided invaluable help in testing code for me on compilers to which I had no access. Without the staff at the Corporate and Professional Publishing Division of Addison-Wesley, there would be no book, and I am indebted to Kim Dawley, Lana Langlois, Simone Payment, Marty Rabinowitz, Pradeepa Siva, John Wait, and the rest of the staff for their encouragement, patience, and help with the production of this work. Chris Guzikowski helped draft the back cover copy for this book, and Tim Johnson stole time from his research on low-temperature physics to critique later versions of that text. Tom Cargill graciously agreed to make his C++ Report article on exceptions available at the Addison-Wesley Internet site. The People Kathy Reed was responsible for my introduction to programming; surely she didn't deserve to have to put up with a kid like me. Donald French had faith in my ability to develop and present C++ teaching materials when I had no track record. He also introduced me to John Wait, my editor at Addison-Wesley, an act for which I will always be grateful. The triumvirate at Beaver Ridge — Jayni Besaw, Lorri Fields, and Beth McKee — provided untold entertainment on my breaks as I worked on the book. My wife, Nancy L. Urbano, put up with me and put up with me and put up with me as I worked on the book, continued to work on the book, and kept working on the book. How many times did she hear me say we'd do something after the book was done? Now the book is done, and we will do those things. She amazes me. I love her. Finally, I must acknowledge our puppy, Persephone, whose existence changed our world forever. Without her, this book would have been finished both sooner and with less sleep deprivation, but also with substantially less comic relief. This page intentionally left blank Introduction These are heady days for C++ programmers. Commercially available less than a decade, C++ has nevertheless emerged as the language of choice for systems programming on nearly all major computing platforms. Companies and individuals with challenging programming problems increasingly embrace the language, and the question faced by those who do not use C++ is often when they will start, not if. Standardization of C++ is complete, and the breadth and scope of the accompanying library — which both dwarfs and subsumes that of C — makes it possible to write rich, complex programs without sacrificing portability or implementing common algorithms and data structures from scratch. C++ compilers continue to proliferate, the features they offer continue to expand, and the quality of the code they generate continues to improve. Tools and environments for C++ development grow ever more abundant, powerful, and robust. Commercial libraries all but obviate the need to write code in many application areas. Introduction As the language has matured and our experience with it has increased, our needs for information about it have changed. In 1990, people wanted to know what C++ was. By 1992, they wanted to know how to make it work. Now C++ programmers ask higher-level questions: How can I design my software so it will adapt to future demands? How can I improve the efficiency of my code without compromising its correctness or making it harder to use? How can I implement sophisticated functionality not directly supported by the language? In this book, I answer these questions and many others like them. This book shows how to design and implement C++ software that is more effective: more likely to behave correctly; more robust in the face of exceptions; more efficient; more portable; makes better use of language features; adapts to change more gracefully; works better in a mixed-language environment; is easier to use correctly; is harder to use incorrectly. In short, software that's just better. The material in this book is divided into 35 Items. Each Item summarizes accumulated wisdom of the C++ programming community on a particular topic. Most Items take the form of guidelines, and the explanation accompanying each guideline describes why the guideline exists, what happens if you fail to follow it, and under what conditions it makes sense to violate the guideline anyway. Items fall into several categories. Some concern particular language features, especially newer features with which you may have little experience. For example, Items 9 through 15 are devoted to exceptions. Other Items explain how to combine the features of the language to achieve higher-level goals. Items 25 through 31, for instance, describe how to constrain the number or placement of objects, how to create functions that act "virtual" on the type of more than one object, how to create "smart pointers," and more. Still other Items address broader topics; Items 16 through 24 focus on efficiency. No matter what the topic of a particular Item, each takes a no-nonsense approach to the subject. In More Effective C++, you learn how to use C++ more effectively. The descriptions of language features that make up the bulk of most C++ texts are in this book mere background information. An implication of this approach is that you should be familiar with C++ before reading this book. I take for granted that you understand classes, protection levels, virtual and nonvirtual functions, etc., and I assume you are acquainted with the concepts behind templates and exceptions. At the same time, I don't expect you to be a language expert, so when poking into lesser-known corners of C++, I always explain what's going on. The C++ in More Effective C++ I describe in this book is the language specified by the 1998 International Standard for C++. This means I may use a few features your compilers don't yet support. Don't worry. The only "new" feature I assume you have is templates, and templates are now almost universally available. I use exceptions, too, but that use is largely confined to Items 9 through 15, which are specifically devoted to exceptions. If you don't have access to a compiler offering exceptions, that's okay. It won't affect your ability to take advantage of the material in the other parts of the book. Furthermore, you should read Items 9 through 15 even if you don't have support for exceptions, because those items examine issues you need to understand in any case. I recognize that just because the standardization committee blesses a feature or endorses a practice, there's no guarantee that the feature is present in current compilers or the practice is applicable to existing environments. When faced with a discrepancy between theory (what the committee says) and practice (what actually works), I discuss both, though my bias is toward things that work. Because I discuss both, this book will aid you as your compilers approach conformance with the standard. It will show you how to use existing constructs to approximate language features your compilers don't yet support, and it will guide you when you decide to transform workarounds into newly-supported features. Notice that I refer to your compilers — plural. Different compilers implement varying approximations to the standard, so I encourage you to develop your code under at least two compilers. Doing so will help you avoid inadvertent dependence on one vendor's proprietary language extension or its misinterpretation of the standard. It will also help keep you away from the bleeding edge of compiler technology, e.g., from new features supported by only one vendor. Such features are often poorly implemented (buggy or slow — frequently both), and upon their introduction, the C++ community lacks experience to advise you in their proper use. Blazing trails can be exciting, but when your goal is producing reliable code, it's often best to let others test the waters before jumping in. There are two constructs you'll see in this book that may not be familiar to you. Both are relatively recent language extensions. Some compilers support them, but if your compilers don't, you can easily approximate them with features you do have. The first construct is the bool type, which has as its values the keywords true and false. If your compilers haven't implemented bool, there are two ways to approximate it. One is to use a global enum: enum bool { false, true }; This allows you to overload functions on the basis of whether they take a bool or an int, but it has the disadvantage that the built-in comparison operators (i.e., ==, <, >=, etc.) still return ints. As a result, code like the following will not behave the way it's supposed to: void f(int); void f(bool); int x, y; ... f(x < y); // calls f(int), but it // should call f(bool) The enum approximation may thus lead to code whose behavior changes when you submit it to a compiler that truly supports bool. An alternative is to use a typedef for bool and constant objects for true and false: typedef int bool; const bool false = 0; const bool true = 1; This is compatible with the traditional semantics of C and C++, and the behavior of programs using this approximation won't change when they're ported to bool-supporting compilers. The drawback is that you can't differentiate between bool and int when overloading functions. Both approximations are reasonable. Choose the one that best fits your circumstances. The second new construct is really four constructs, the casting forms static_cast, const_cast, dynamic_cast, and reinterpret_cast. If you're not familiar with these casts, you'll want to turn to Item 2 and read all about them. Not only do they do more than the C-style casts they replace, they do it better. I use these new casting forms whenever I need to perform casts in this book. There is more to C++ than the language itself. There is also the standard library. Where possible, I employ the standard string type instead of using raw char* pointers, and I encourage you to do the same. string objects are no more difficult to manipulate than char*based strings, and they relieve you of most memory-management concerns. Furthermore, string objects are less susceptible to memory leaks if an exception is thrown (see Items 9 and 10). A well-implemented string type can hold its own in an efficiency contest with its char* equivalent, and it may even do better. (For insight into how this could be, see Item 29.) If you don't have access to an implementation of the standard string type, you almost certainly have access to some string-like class. Use it. Just about anything is preferable to raw char*s. I use data structures from the standard library whenever I can. Such data structures are drawn from the Standard Template Library (the "STL" — see Item 35). The STL includes bitsets, vectors, lists, queues, stacks, maps, sets, and more, and you should prefer these standardized data structures to the ad hoc equivalents you might otherwise be tempted to write. Your compilers may not have the STL bundled in, but don't let that keep you from using it. Thanks to Silicon Graphics, you can download a free copy that works with many compilers from the SGI STL web site: . Introduction 5 If you currently use a library of algorithms and data structures and are happy with it, there's no need to switch to the STL just because it's "standard." However, if you have a choice between using an STL component or writing your own code from scratch, you should lean toward using the STL. Remember code reuse? STL (and the rest of the standard library) has lots of code that is very much worth reusing. Conventions and Terminology Any time I mention inheritance in this book, I mean public inheritance. If I don't mean public inheritance, I'll say so explicitly. When drawing inheritance hierarchies, I depict base-derived relationships by drawing arrows from derived classes to base classes. For example, here is a hierarchy from Item 31: GameObject SpaceShip SpaceStation Asteroid This notation is the reverse of the convention I employed in the first (but not the second) edition of Effective C++. I'm now convinced that most C++ practitioners draw inheritance arrows from derived to base classes, and I am happy to follow suit. Within such diagrams, abstract classes (e.g., GameObject) are shaded and concrete classes (e.g., SpaceShip) are unshaded. Inheritance gives rise to pointers and references with two different types, a static type and a dynamic type. The static type of a pointer or reference is its declared type. The dynamic type is determined by the type of object it actually refers to. Here are some examples based on the classes above: GameObject *pgo = new SpaceShip; // static type of pgo is // GameObject*, dynamic // type is SpaceShip* Asteroid *pa = new Asteroid; // static type of pa is // Asteroid*. So is its // dynamic type pgo = pa; // // // // static type of pgo is still (and always) GameObject*. Its dynamic type is now Asteroid* 6 Introduction GameObject& rgo = *pa; // static type of rgo is // GameObject, dynamic // type is Asteroid These examples also demonstrate a naming convention I like. pgo is a pointer-to-GameObject; pa is a pointer-to-Asteroid; rgo is a reference-to-GameObject. I often concoct pointer and reference names in this fashion. Two of my favorite parameter names are lhs and rhs, abbreviations for "left-hand side" and "right-hand side," respectively. To understand the rationale behind these names, consider a class for representing rational numbers: class Rational { ... }; If I wanted a function to compare pairs of Rational objects, I'd declare it like this: bool operator==(const Rational& lhs, const Rational& rhs); That would let me write this kind of code: Rational r1, r2; ... if (r1 == r2) ... Within the call to operator==, r1 appears on the left-hand side of the "==" and is bound to lhs, while r2 appears on the right-hand side of the "==" and is bound to rhs. Other abbreviations I employ include ctor for "constructor," dtor for "destructor," and RTTI for C++'s support for runtime type identification (of which dynamic_cast is the most commonly used component). When you allocate memory and fail to free it, you have a memory leak. Memory leaks arise in both C and C++, but in C++, memory leaks leak more than just memory. That's because C++ automatically calls constructors when objects are created, and constructors may themselves allocate resources. For example, consider this code: class Widget { ... }; // some class — it doesn't // matter what it is Widget *pw = new Widget; // dynamically allocate a // Widget object ... // assume pw is never // deleted This code leaks memory, because the Widget pointed to by pw is never deleted. However, if the Widget constructor allocates additional re- Introduction 7 sources that are to be released when the Widget is destroyed (such as file descriptors, semaphores, window handles, database locks, etc.), those resources are lost just as surely as the memory is. To emphasize that memory leaks in C++ often leak other resources, too, I usually speak of resource leaks in this book rather than memory leaks. You won't see many inline functions in this book. That's not because I dislike inlining. Far from it, I believe that inline functions are an important feature of C++. However, the criteria for determining whether a function should be inlined can be complex, subtle, and platform-dependent. As a result, I avoid inlining unless there is a point about inlining I wish to make. When you see a non-inline function in More Effective C++, that doesn't mean I think it would be a bad idea to declare the function inline, it just means the decision to inline that function is independent of the material I'm examining at that point in the book. A few C++ features have been deprecated by the standardization committee. Such features are slated for eventual removal from the language, because newer features have been added that do what the deprecated features do, but do it better. In this book, I identify deprecated constructs and explain what features replace them. You should try to avoid deprecated features where you can, but there's no reason to be overly concerned about their use. In the interest of preserving backward compatibility for their customers, compiler vendors are likely to support deprecated features for many years. A client is somebody (a programmer) or something (a class or function, typically) that uses the code you write. For example, if you write a Date class (for representing birthdays, deadlines, when the Second Coming occurs, etc.), anybody using that class is your client. Furthermore, any sections of code that use the Date class are your clients as well. Clients are important. In fact, clients are the name of the game! If nobody uses the software you write, why write it? You will find I worry a lot about making things easier for clients, often at the expense of making things more difficult for you, because good software is "clientcentric" — it revolves around clients. If this strikes you as unreasonably philanthropic, view it instead through a lens of self-interest. Do you ever use the classes or functions you write? If so, you're your own client, so making things easier for clients in general also makes them easier for you. When discussing class or function templates and the classes or functions generated from them, I reserve the right to be sloppy about the difference between the templates and their instantiations. For example, if Array is a class template taking a type parameter T, I may refer to a particular instantiation of the template as an Array, even though 8 Introduction Array is really the name of the class. Similarly, if swap is a function template taking a type parameter T, I may refer to an instantiation as swap instead of swap. In cases where this kind of shorthand might be unclear, I include template parameters when referring to template instantiations. Reporting Bugs, Making Suggestions, Getting Book Updates I have tried to make this book as accurate, readable, and useful as possible, but I know there is room for improvement. If you find an error of any kind — technical, grammatical, typographical, whatever — please tell me about it. I will try to correct the mistake in future printings of the book, and if you are the first person to report it, I will gladly add your name to the book's acknowledgments. If you have other suggestions for improvement, I welcome those, too. I continue to collect guidelines for effective programming in C++. If you have ideas for new guidelines, I'd be delighted if you'd share them with me. Send your guidelines, your comments, your criticisms, and your bug reports to: Scott Meyers c/o Editor-in-Chief, Corporate and Professional Publishing Addison-Wesley Publishing Company 1 Jacob Way Reading, MA 01867 U. S. A. Alternatively, you may send electronic mail to I maintain a list of changes to this book since its first printing, including bug-fixes, clarifications, and technical updates. This list, along with other book-related information, is available from Addison-Wesley at World Wide Web URL +.html. It is also available via anonymous FTP from ftp.awl.com in the directory cp/mec++. If you would like a copy of the list of changes to this book, but you lack access to the Internet, please send a request to one of the addresses above, and I will see that the list is sent to you. If you'd like to be notified when I make changes to this book, consider joining my mailing list. For details, consult MailingList/index.html. Enough preliminaries. On with the show! Basics Ah, the basics. Pointers, references, casts, arrays, constructors — you can't get much more basic than that. All but the simplest C++ programs use most of these features, and many programs use them all. Basics In spite of our familiarity with these parts of the language, sometimes they can still surprise us. This is especially true for programmers making the transition from C to C++, because the concepts behind references, dynamic casts, default constructors, and other non-C features are usually a little murky. This chapter describes the differences between pointers and references and offers guidance on when to use each. It introduces the new C++ syntax for casts and explains why the new casts are superior to the Cstyle casts they replace. It examines the C notion of arrays and the C++ notion of polymorphism, and it describes why mixing the two is an idea whose time will never come. Finally, it considers the pros and cons of default constructors and suggests ways to work around language restrictions that encourage you to have one when none makes sense. By heeding the advice in the items that follow, you'll make progress toward a worthy goal: producing software that expresses your design intentions clearly and correctly. Item 1: Distinguish between pointers and references. Pointers and references look different enough (pointers use the "*" and "->" operators, references use "."), but they seem to do similar things. Both pointers and references let you refer to other objects indirectly. How, then, do you decide when to use one and not the other? Pointers versus References First, recognize that there is no such thing as a null reference. A reference must always refer to some object. As a result, if you have a variable whose purpose is to refer to another object, but it is possible that there might not be an object to refer to, you should make the variable 10 Item 1 a pointer, because then you can set it to null. On the other hand, if the variable must always refer to an object, i.e., if your design does not allow for the possibility that the variable is null, you should probably make the variable a reference. "But wait," you wonder, "what about underhandedness like this?" char *pc = 0; // set pointer to null char& rc = *pc; // make reference refer to // dereferenced null pointer Well, this is evil, pure and simple. The results are undefined (compilers can generate output to do anything they like), and people who write this kind of code should be shunned until they agree to cease and desist. If you have to worry about things like this in your software, you're probably best off avoiding references entirely. Either that or finding a better class of programmers to work with. We'll henceforth ignore the possibility that a reference can be "null." Because a reference must refer to an object, C++ requires that references be initialized: string& rs; // error! References must // be initialized string s("xyzzy"); string& rs = s; // okay, rs refers to s Pointers are subject to no such restriction: string *ps; // uninitialized pointer: // valid but risky The fact that there is no such thing as a null reference implies that it can be more efficient to use references than to use pointers. That's because there's no need to test the validity of a reference before using it: void printDouble(const double& rd) { cout ~string(); // call the object's dtor operator delete(ps); // deallocate the memory // the object occupied One implication of this is that if you want to deal only with raw, uninitialized memory, you should bypass the new and delete operators entirely. Instead, you should call operator new to get the memory and operator delete to return it to the system: void *buffer = operator new(50*sizeof(char)); // allocate enough // memory to hold 50 // chars; call no ctors ... operator delete(buffer); // deallocate the memory; // call no dtors 42 Item 8 This is the C++ equivalent of calling malloc and free. If you use placement new to create an object in some memory, you should avoid using the delete operator on that memory. That's because the delete operator calls operator delete to deallocate the memory, but the memory containing the object wasn't allocated by operator new in the first place; placement new just returned the pointer that was passed to it. Who knows where that pointer came from? Instead, you should undo the effect of the constructor by explicitly calling the object's destructor: // functions for allocating and deallocating memory in // shared memory void * mallocShared(size_t size); void freeShared(void *memory); void *sharedMemory = mallocShared(sizeof(Widget)); Widget *pw = // as above, constructWidgetInBuffer(sharedMemory, 10); // placement // new is used ... delete pw; // undefined! sharedMemory came from // mallocShared, not operator new pw->~Widget(); // fine, destructs the Widget pointed to // by pw, but doesn't deallocate the // memory containing the Widget freeShared(pw); // fine, deallocates the memory pointed // to by pw, but calls no destructor As this example demonstrates, if the raw memory passed to placement new was itself dynamically allocated (through some unconventional means), you must still deallocate that memory if you wish to avoid a memory leak. Arrays So far so good, but there's farther to go. Everything we've examined so far concerns itself with only one object at a time. What about array allocation? What happens here? string *ps = new string[10]; // allocate an array of // objects The new being used is still the new operator, but because an array is being created, the new operator behaves slightly differently from the case of single-object creation. For one thing, memory is no longer allocated by operator new. Instead, it's allocated by the array-allocation equivalent, a function called operator new[] (often referred to as "ar- The Different Meanings of new and delete 43 ray new.") Like operator new, operator new[] can be overloaded. This allows you to seize control of memory allocation for arrays in the same way you can control memory allocation for single objects. (operator new[] is a relatively recent addition to C++, so your compilers may not support it yet. If they don't, the global version of operator new will be used to allocate memory for every array, regardless of the type of objects in the array. Customizing array-memory allocation under such compilers is daunting, because it requires that you rewrite the global operator new. This is not a task to be undertaken lightly. By default, the global operator new handles all dynamic memory allocation in a program, so any change in its behavior has a dramatic and pervasive effect. Furthermore, there is only one global operator new with the "normal" signature (i.e., taking the single size_t parameter), so if you decide to claim it as your own, you instantly render your software incompatible with any library that makes the same decision. (See also Item 27.) As a result of these considerations, custom memory management for arrays is not usually a reasonable design decision for compilers lacking support for operator new[].) The second way in which the new operator behaves differently for arrays than for objects is in the number of constructor calls it makes. For arrays, a constructor must be called for each object in the array: string *ps = new string[10]; // // // // call operator memory for 10 then call the ctor for each new[] to allocate string objects, default string array element Similarly, when the delete operator is used on an array, it calls a destructor for each array element and then calls operator delete[] to deallocate the memory: delete [] ps; // // // // call the string dtor for each array element, then call operator delete[] to deallocate the array's memory Just as you can replace or overload operator delete, you can replace or overload operator delete[]. There are some restrictions on how they can be overloaded, however; consult a good C++ text for details. (For ideas on good C++ texts, see the recommendations beginning on page 285.) So there you have it. The new and delete operators are built-in and beyond your control, but the memory allocation and deallocation functions they call are not. When you think about customizing the behavior of the new and delete operators, remember that you can't really do it. You can modify how they do what they do, but what they do is fixed by the language. Exceptions The addition of exceptions to C++ changes things. Profoundly. Radically. Possibly uncomfortably. The use of raw, unadorned pointers, for example, becomes risky. Opportunities for resource leaks increase in number. It becomes more difficult to write constructors and destructors that behave the way we want them to. Special care must be taken to prevent program execution from abruptly halting. Executables and libraries typically increase in size and decrease in speed. Exceptions And these are just the things we know. There is much the C++ community does not know about writing programs using exceptions, including, for the most part, how to do it correctly. There is as yet no agreement on a body of techniques that, when applied routinely, leads to software that behaves predictably and reliably when exceptions are thrown. (For insight into some of the issues involved, see the article by Tom Cargill I refer to on page 287.) We do know this much: programs that behave well in the presence of exceptions do so because they were designed to, not because they happen to. Exception-safe programs are not created by accident. The chances of a program behaving well in the presence of exceptions when it was not designed for exceptions are about the same as the chances of a program behaving well in the presence of multiple threads of control when it was not designed for multi-threaded execution: about zero. That being the case, why use exceptions? Error codes have sufficed for C programmers ever since C was invented, so why mess with exceptions, especially if they're as problematic as I say? The answer is simple: exceptions cannot be ignored. If a function signals an exceptional condition by setting a status variable or returning an error code, there is no way to guarantee the function's caller will check the variable or examine the code. As a result, execution may continue long past the point where the condition was encountered. If the function signals the Using Destructors to Prevent Resource Leaks 45 condition by throwing an exception, however, and that exception is not caught, program execution immediately ceases. This is behavior that C programmers can approach only by using setjmp and longjmp. But longjmp exhibits a serious deficiency when used with C++: it fails to call destructors for local objects when it adjusts the stack. Most C++ programs depend on such destructors being called, so setjmp and longjmp make a poor substitute for true exceptions. If you need a way of signaling exceptional conditions that cannot be ignored, and if you must ensure that local destructors are called when searching the stack for code that can handle exceptional conditions, you need C++ exceptions. It's as simple as that. Because we have much to learn about programming with exceptions, the Items that follow comprise an incomplete guide to writing exception-safe software. Nevertheless, they introduce important considerations for anyone using exceptions in C++. By heeding the guidance in the material below, you'll improve the correctness, robustness, and efficiency of the software you write, and you'll sidestep many problems that commonly arise when working with exceptions. Item 9: Use destructors to prevent resource leaks. Say good-bye to pointers. Admit it: you never really liked them that much anyway. Using Destructors to Prevent Resource Leaks Okay, you don't have to say good-bye to all pointers, but you do need to say sayonara to pointers that are used to manipulate local resources. Suppose, for example, you're writing software at the Shelter for Adorable Little Animals, an organization that finds homes for puppies and kittens. Each day the shelter creates a file containing information on the adoptions it arranged that day, and your job is to write a program to read these files and do the appropriate processing for each adoption. A reasonable approach to this task is to define an abstract base class, ALA ("Adorable Little Animal"), plus concrete derived classes for puppies and kittens. A virtual function, processAdoption, handles the necessary species-specific processing: ALA Puppy Kitten 46 Item 9 class ALA { public: virtual void processAdoption() = 0; ... }; class Puppy: public ALA { public: virtual void processAdoption(); ... }; class Kitten: public ALA { public: virtual void processAdoption(); ... }; You'll need a function that can read information from a file and produce either a Puppy object or a Kitten object, depending on the information in the file. This is a perfect job for a virtual constructor, a kind of function described in Item 25. For our purposes here, the function's declaration is all we need: // read animal information from s, then return a pointer // to a newly allocated object of the appropriate type ALA * readALA(istream& s); The heart of your program is likely to be a function that looks something like this: void processAdoptions(istream& dataSource) { while (dataSource) { // while there's data ALA *pa = readALA(dataSource); // get next animal pa->processAdoption(); // process adoption delete pa; // delete object that } // readALA returned } This function loops through the information in dataSource, processing each entry as it goes. The only mildly tricky thing is the need to remember to delete pa at the end of each iteration. This is necessary because readALA creates a new heap object each time it's called. Without the call to delete, the loop would contain a resource leak. Now consider what would happen if pa->processAdoption threw an exception. processAdoptions fails to catch exceptions, so the exception would propagate to processAdoptions's caller. In doing so, all statements in processAdoptions after the call to pa->processAdoption would be skipped, and that means pa would never be deleted. As Using Destructors to Prevent Resource Leaks 47 a result, anytime pa->processAdoption throws an exception, processAdoptions contains a resource leak. Plugging the leak is easy enough, void processAdoptions(istream& dataSource) { while (dataSource) { ALA *pa = readALA(dataSource); try { pa->processAdoption(); } catch (...) { // catch all exceptions delete pa; // avoid resource leak when an // exception is thrown throw; // propagate exception to caller } delete pa; } // avoid resource leak when no // exception is thrown } but then you have to litter your code with try and catch blocks. More importantly, you are forced to duplicate cleanup code that is common to both normal and exceptional paths of control. In this case, the call to delete must be duplicated. Like all replicated code, this is annoying to write and difficult to maintain, but it also feels wrong. Regardless of whether we leave processAdoptions by a normal return or by throwing an exception, we need to delete pa, so why should we have to say that in more than one place? We don't have to if we can somehow move the cleanup code that must always be executed into the destructor for an object local to processAdoptions. That's because local objects are always destroyed when leaving a function, regardless of how that function is exited. (The only exception to this rule is when you call longjmp, and this shortcoming of longjmp is the primary reason why C++ has support for exceptions in the first place.) Our real concern, then, is moving the delete from processAdoptions into a destructor for an object local to processAdoptions. The solution is to replace the pointer pa with an object that acts like a pointer. That way, when the pointer-like object is (automatically) destroyed, we can have its destructor call delete. Objects that act like pointers, but do more, are called smart pointers, and, as Item 28 explains, you can make pointer-like objects very smart indeed. In this case, we don't need a particularly brainy pointer, we just need a 48 Item 9 pointer-like object that knows enough to delete what it points to when the pointer-like object goes out of scope. It's not difficult to write a class for such objects, but we don't need to. The standard C++ library contains a class template called auto_ptr that does just what we want. Each auto_ptr class takes a pointer to a heap object in its constructor and deletes that object in its destructor. Boiled down to these essential functions, auto_ptr looks like this: template class auto_ptr { public: auto_ptr(T *p = 0): ptr(p) {} ~auto_ptr() { delete ptr; } private: T *ptr; }; // save ptr to object // delete ptr to object // raw ptr to object The standard version of auto_ptr is much fancier, and this strippeddown implementation isn't suitable for real use† (we must add at least the copy constructor, assignment operator, and pointer-emulating functions discussed in Item 28), but the concept behind it should be clear: use auto_ptr objects instead of raw pointers, and you won't have to worry about heap objects not being deleted, not even when exceptions are thrown. (Because the auto_ptr destructor uses the single-object form of delete, auto_ptr is not suitable for use with pointers to arrays of objects. If you'd like an auto_ptr-like template for arrays, you'll have to write your own. In such cases, however, it's often a better design decision to use a vector instead of an array, anyway.) Using an auto_ptr object instead of a raw pointer, processAdoptions looks like this: void processAdoptions(istream& dataSource) { while (dataSource) { auto_ptr pa(readALA(dataSource)); pa->processAdoption(); } } This version of processAdoptions differs from the original in only two ways. First, pa is declared to be an auto_ptr object, not a raw ALA* pointer. Second, there is no delete statement at the end of the loop. That's it. Everything else is identical, because, except for destruction, auto_ptr objects act just like normal pointers. Easy, huh? † A complete implementation of an almost-standard auto_ptr appears on pages 291-294. Using Destructors to Prevent Resource Leaks 49 The idea behind auto_ptr — using an object to store a resource that needs to be automatically released and relying on that object's destructor to release it — applies to more than just pointer-based resources. Consider a function in a GUI application that needs to create a window to display some information: // this function may leak resources if an exception // is thrown void displayInfo(const Information& info) { WINDOW_HANDLE w(createWindow()); display info in window corresponding to w; destroyWindow(w); } Many window systems have C-like interfaces that use functions like createWindow and destroyWindow to acquire and release window resources. If an exception is thrown during the process of displaying info in w, the window for which w is a handle will be lost just as surely as any other dynamically allocated resource. The solution is the same as it was before. Create a class whose constructor and destructor acquire and release the resource: // class for acquiring and releasing a window handle class WindowHandle { public: WindowHandle(WINDOW_HANDLE handle): w(handle) {} ~WindowHandle() { destroyWindow(w); } operator WINDOW_HANDLE() { return w; } // see below private: WINDOW_HANDLE w; // The following functions are declared private to prevent // multiple copies of a WINDOW_HANDLE from being created. // See Item 28 for a discussion of a more flexible approach. WindowHandle(const WindowHandle&); WindowHandle& operator=(const WindowHandle&); }; This looks just like the auto_ptr template, except that assignment and copying are explicitly prohibited, and there is an implicit conversion operator that can be used to turn a WindowHandle into a WINDOW_HANDLE. This capability is essential to the practical application of a WindowHandle object, because it means you can use a WindowHandle just about anywhere you would normally use a raw WINDOW_HANDLE. (See Item 5, however, for why you should generally be leery of implicit type conversion operators.) 50 Item 10 Given the WindowHandle class, we can rewrite displayInfo as follows: // this function avoids leaking resources if an // exception is thrown void displayInfo(const Information& info) { WindowHandle w(createWindow()); display info in window corresponding to w; } Even if an exception is thrown within displayInfo, the window created by createWindow will always† be destroyed. By adhering to the rule that resources should be encapsulated inside objects, you can usually avoid resource leaks in the presence of exceptions. But what happens if an exception is thrown while you're in the process of acquiring a resource, e.g., while you're in the constructor of a resource-acquiring class? What happens if an exception is thrown during the automatic destruction of such resources? Don't constructors and destructors call for special techniques? They do, and you can read about them in Items 10 and 11. Item 10: Prevent resource leaks in constructors. Imagine you're developing software for a multimedia address book. Such an address book might hold, in addition to the usual textual information of a person's name, address, and phone numbers, a picture of the person and the sound of their voice (possibly giving the proper pronunciation of their name). Avoiding Resource Leaks in Constructors To implement the book, you might come up with a design like this: class Image { // for image data public: Image(const string& imageDataFileName); ... }; class AudioClip { // for audio data public: AudioClip(const string& audioDataFileName); ... }; class PhoneNumber { ... }; // for holding phone numbers † Well, almost always. If the exception is not caught, the program will terminate. In that case, there is no guarantee that local objects (such as w in the example) will have their destructors called. Some compilers call them, some do not. Both behaviors are valid. Avoiding Resource Leaks in Constructors class BookEntry { public: BookEntry(const const const const ~BookEntry(); 51 // for each entry in the // address book string& string& string& string& name, address = "", imageFileName = "", audioClipFileName = ""); // phone numbers are added via this function void addPhoneNumber(const PhoneNumber& number); ... private: string theName; string theAddress; list thePhones; Image *theImage; AudioClip *theAudioClip; }; // // // // // person's name their address their phone numbers their image an audio clip from them Each BookEntry must have name data, so you require that as a constructor argument (see Item 4), but the other fields — the person's address and the names of files containing image and audio data — are optional. Note the use of the list class to hold the person's phone numbers. This is one of several container classes that are part of the standard C++ library (see Item 35). A straightforward way to write the BookEntry constructor and destructor is as follows: BookEntry::BookEntry(const string& name, const string& address, const string& imageFileName, const string& audioClipFileName) : theName(name), theAddress(address), theImage(0), theAudioClip(0) { if (imageFileName != "") { theImage = new Image(imageFileName); } if (audioClipFileName != "") { theAudioClip = new AudioClip(audioClipFileName); } } BookEntry::~BookEntry() { delete theImage; delete theAudioClip; } 52 Item 10 The constructor initializes the pointers theImage and theAudioClip to null, then makes them point to real objects if the corresponding arguments are non-empty strings. The destructor deletes both pointers, thus ensuring that a BookEntry object doesn't give rise to a resource leak. Because C++ guarantees it's safe to delete null pointers, BookEntry's destructor need not check to see if the pointers actually point to something before deleting them. Everything looks fine here, and under normal conditions everything is fine, but under abnormal conditions — under exceptional conditions — things are not fine at all. Consider what will happen if an exception is thrown during execution of this part of the BookEntry constructor: if (audioClipFileName != "") { theAudioClip = new AudioClip(audioClipFileName); } An exception might arise because operator new (see Item 8) is unable to allocate enough memory for an AudioClip object. One might also arise because the AudioClip constructor itself throws an exception. Regardless of the cause of the exception, if one is thrown within the BookEntry constructor, it will be propagated to the site where the BookEntry object is being created. Now, if an exception is thrown during creation of the object theAudioClip is supposed to point to (thus transferring control out of the BookEntry constructor), who deletes the object that theImage already points to? The obvious answer is that BookEntry's destructor does, but the obvious answer is wrong. BookEntry's destructor will never be called. Never. C++ destroys only fully constructed objects, and an object isn't fully constructed until its constructor has run to completion. So if a BookEntry object b is created as a local object, void testBookEntryClass() { BookEntry b("Addison-Wesley Publishing Company", "One Jacob Way, Reading, MA 01867"); ... } and an exception is thrown during construction of b, b's destructor will not be called. Furthermore, if you try to take matters into your own hands by allocating b on the heap and then calling delete if an exception is thrown, Avoiding Resource Leaks in Constructors 53 void testBookEntryClass() { BookEntry *pb = 0; try { pb = new BookEntry("Addison-Wesley Publishing Company", "One Jacob Way, Reading, MA 01867"); ... } catch (...) { // catch all exceptions delete pb; // delete pb when an exception is thrown throw; // propagate exception to // caller delete pb; // delete pb normally } you'll find that the Image object allocated inside BookEntry's constructor is still lost, because no assignment is made to pb unless the new operation succeeds. If BookEntry's constructor throws an exception, pb will be the null pointer, so deleting it in the catch block does nothing except make you feel better about yourself. Using the smart pointer class auto_ptr (see Item 9) instead of a raw BookEntry* won't do you any good either, because the assignment to pb still won't be made unless the new operation succeeds. There is a reason why C++ refuses to call destructors for objects that haven't been fully constructed, and it's not simply to make your life more difficult. It's because it would, in many cases, be a nonsensical thing — possibly a harmful thing — to do. If a destructor were invoked on an object that wasn't fully constructed, how would the destructor know what to do? The only way it could know would be if bits had been added to each object indicating how much of the constructor had been executed. Then the destructor could check the bits and (maybe) figure out what actions to take. Such bookkeeping would slow down constructors, and it would make each object larger, too. C++ avoids this overhead, but the price you pay is that partially constructed objects aren't automatically destroyed. Because C++ won't clean up after objects that throw exceptions during construction, you must design your constructors so that they clean up after themselves. Often, this involves simply catching all possible exceptions, executing some cleanup code, then rethrowing the exception so it continues to propagate. This strategy can be incorporated into the BookEntry constructor like this: 54 Item 10 BookEntry::BookEntry(const string& name, const string& address, const string& imageFileName, const string& audioClipFileName) : theName(name), theAddress(address), theImage(0), theAudioClip(0) { try { // this try block is new if (imageFileName != "") { theImage = new Image(imageFileName); } if (audioClipFileName != "") { theAudioClip = new AudioClip(audioClipFileName); } } catch (...) { // catch any exception delete theImage; // perform necessary delete theAudioClip; // cleanup actions throw; // propagate the exception } } There is no need to worry about BookEntry's non-pointer data members. Data members are automatically initialized before a class's constructor is called, so if a BookEntry constructor body begins executing, the object's theName, theAddress, and thePhones data members have already been fully constructed. As fully constructed objects, these data members will be automatically destroyed even if an exception arises in the BookEntry constructor†. Of course, if these objects' constructors call functions that might throw exceptions, those constructors have to worry about catching the exceptions and performing any necessary cleanup before allowing them to propagate. You may have noticed that the statements in BookEntry's catch block are almost the same as those in BookEntry's destructor. Code duplication here is no more tolerable than it is anywhere else, so the best way to structure things is to move the common code into a private helper function and have both the constructor and the destructor call it: class BookEntry { public: ... // as before private: ... void cleanup(); // common cleanup statements }; void BookEntry::cleanup() { delete theImage; delete theAudioClip; } BookEntry::BookEntry(const string& name, const string& address, const string& imageFileName, const string& audioClipFileName) : theName(name), theAddress(address), theImage(0), theAudioClip(0) { try { ... // as before } catch (...) { cleanup(); // release resources throw; // propagate exception } } BookEntry::~BookEntry() { cleanup(); } This is nice, but it doesn't put the topic to rest. Let us suppose we design our BookEntry class slightly differently so that theImage and theAudioClip are constant pointers: class BookEntry { public: ... // as above private: ... Image * const theImage; // pointers are now AudioClip * const theAudioClip; // const } † Provided, again, that the exception is caught. Item 10 55 Such pointers must be initialized via the member initialization lists of BookEntry's constructors, because there is no other way to give const pointers a value. A common temptation is to initialize theImage and theAudioClip like this, 56 Item 10 // an implementation that may leak resources if an // exception is thrown BookEntry::BookEntry(const string& name, const string& address, const string& imageFileName, const string& audioClipFileName) : theName(name), theAddress(address), theImage(imageFileName != "" ? new Image(imageFileName) : 0), theAudioClip(audioClipFileName != "" ? new AudioClip(audioClipFileName) : 0) {} but this leads to the problem we originally wanted to eliminate: if an exception is thrown during initialization of theAudioClip, the object pointed to by theImage is never destroyed. Furthermore, we can't solve the problem by adding try and catch blocks to the constructor, because try and catch are statements, and member initialization lists allow only expressions. (That's why we had to use the ?: syntax instead of the if-then-else syntax in the initialization of theImage and theAudioClip.) Nevertheless, the only way to perform cleanup chores before exceptions propagate out of a constructor is to catch those exceptions, so if we can't put try and catch in a member initialization list, we'll have to put them somewhere else. One possibility is inside private member functions that return pointers with which theImage and theAudioClip should be initialized: class BookEntry { public: ... // as above private: ... // data members as above Image * initImage(const string& imageFileName); AudioClip * initAudioClip(const string& audioClipFileName); }; BookEntry::BookEntry(const string& name, const string& address, const string& imageFileName, const string& audioClipFileName) : theName(name), theAddress(address), theImage(initImage(imageFileName)), theAudioClip(initAudioClip(audioClipFileName)) {} // theImage is initialized first, so there is no need to // worry about a resource leak if this initialization // fails. This function therefore handles no exceptions Image * BookEntry::initImage(const string& imageFileName) { if (imageFileName != "") return new Image(imageFileName); else return 0; } // theAudioClip is initialized second, so it must make // sure theImage's resources are released if an // exception is thrown during initialization of theAudioClip. That's // why this function uses try...catch. AudioClip * BookEntry::initAudioClip(const string& audioClipFileName) { try { if (audioClipFileName != "") { return new AudioClip(audioClipFileName); } else return 0; } catch (...) { delete theImage; throw; } } This is perfectly kosher, and it even solves the problem we've been laboring to overcome. The drawback is that code that conceptually belongs in a constructor is now dispersed across several functions, and that's a maintenance headache. A better solution is to adopt the advice of Item 9 and treat the objects pointed to by theImage and theAudioClip as resources to be managed by local objects. This solution takes advantage of the facts that both theImage and theAudioClip are pointers to dynamically allocated objects and that those objects should be deleted when the pointers themselves go away. This is precisely the set of conditions for which the auto_ptr classes (see Item 9) were designed. We can therefore change the raw pointer types of theImage and theAudioClip to their auto_ptr equivalents: class BookEntry { public: ... // as above private: ... const auto_ptr<Image> theImage; // these are now const auto_ptr<AudioClip> theAudioClip; // auto_ptr objects }; 58 Item 11 Doing this makes BookEntry's constructor leak-safe in the presence of exceptions, and it lets us initialize theImage and theAudioClip using the member initialization list: BookEntry::BookEntry(const string& name, const string& address, const string& imageFileName, const string& audioClipFileName) : theName(name), theAddress(address), theImage(imageFileName != "" ? new Image(imageFileName) : 0), theAudioClip(audioClipFileName != "" ? new AudioClip(audioClipFileName) : 0) {} In this design, if an exception is thrown during initialization of theAudioClip, theImage is already a fully constructed object, so it will automatically be destroyed, just as theName, theAddress, and thePhones. Furthermore, because theImage and theAudioClip are now objects, they'll be destroyed automatically when the BookEntry object containing them is. Hence there's no need to manually delete what they point to. That simplifies BookEntry's destructor considerably: BookEntry::~BookEntry() {} // nothing to do! This means you could eliminate BookEntry's destructor entirely. It all adds up to this: if you replace pointer class members with their corresponding auto_ptr objects, you fortify your constructors against resource leaks in the presence of exceptions, you eliminate the need to manually deallocate resources in destructors, and you allow const member pointers to be handled in the same graceful fashion as non-const pointers. Dealing with the possibility of exceptions during construction can be tricky, but auto_ptr (and auto_ptr-like classes) can eliminate most of the drudgery. Their use leaves behind code that's not only easy to understand, it's robust in the face of exceptions, too. Item 11: Prevent exceptions from leaving destructors. There are two situations in which a destructor is called. The first is when an object is destroyed under "normal" conditions, e.g., when it goes out of scope or is explicitly deleted. The second is when an object is destroyed by the exception-handling mechanism during the stack-unwinding part of exception propagation. Exceptions and Destructors Exceptions and Destructors 59 That being the case, an exception may or may not be active when a destructor is invoked. Regrettably, there is no way to distinguish between these conditions from inside a destructor.† As a result, you must write your destructors under the conservative assumption that an exception is active, because if control leaves a destructor due to an exception while another exception is active, C++ calls the terminate function. That function does just what its name suggests: it terminates execution of your program. Furthermore, it terminates it immediately; not even local objects are destroyed. As an example, consider a Session class for monitoring on-line computer sessions, i.e., things that happen from the time you log in through the time you log out. Each Session object notes the date and time of its creation and destruction: class Session { public: Session(); ~Session(); ... private: static void logCreation(Session *objAddr); static void logDestruction(Session *objAddr); }; The functions logCreation and logDestruction are used to record object creations and destructions, respectively. We might therefore expect that we could code Session's destructor like this: Session::~Session() { logDestruction(this); } This looks fine, but consider what would happen if logDestruction throws an exception. The exception would not be caught in Session's destructor, so it would be propagated to the caller of that destructor. But if the destructor was itself being called because some other exception had been thrown, the terminate function would automatically be invoked, and that would stop your program dead in its tracks. In many cases, this is not what you'll want to have happen. It may be unfortunate that the Session object's destruction can't be logged, it might even be a major inconvenience, but is it really so horrific a prospect that the program can't continue running? If not, you'll have to prevent the exception thrown by logDestruction from propagating out of Session's destructor. The only way to do that is by using try and catch blocks. A naive attempt might look like this, Session::~Session() { try { logDestruction(this); } catch (...) { cerr << "Unable to log destruction of Session object " << "at address " << this << ".\n"; } } † Now there is. In July 1995, the ISO/ANSI standardization committee for C++ added a function, uncaught_exception, that returns true if an exception is active and has not yet been caught. 60 Item 11 but this is probably no safer than our original code. If one of the calls to operator<< in the catch block results in an exception being thrown, we're back where we started, with an exception leaving the Session destructor. We could always put a try block inside the catch block, but that seems a bit extreme. Instead, we'll just forget about logging Session destructions if logDestruction throws an exception: Session::~Session() { try { logDestruction(this); } catch (...) { } } The catch block appears to do nothing, but appearances can be deceiving. That block prevents exceptions thrown from logDestruction from propagating beyond Session's destructor. That's all it needs to do. We can now rest easy knowing that if a Session object is destroyed as part of stack unwinding, terminate will not be called. There is a second reason why it's bad practice to allow exceptions to propagate out of destructors. If an exception is thrown from a destructor and is not caught there, that destructor won't run to completion. (It will stop at the point where the exception is thrown.) If the destructor doesn't run to completion, it won't do everything it's supposed to do. For example, consider a modified version of the Session class where the creation of a session starts a database transaction and the termination of a session ends that transaction: Session::Session() // to keep things simple, { // this ctor handles no // exceptions logCreation(this); startTransaction(); // start DB transaction } Session::~Session() { logDestruction(this); endTransaction(); // end DB transaction } Here, if logDestruction throws an exception, the transaction started in the Session constructor will never be ended. In this case, we might be able to reorder the function calls in Session's destructor to eliminate the problem, but if endTransaction might throw an exception, we've no choice but to revert to try and catch blocks. We thus find ourselves with two good reasons for keeping exceptions from propagating out of destructors. First, it prevents terminate from being called during the stack-unwinding part of exception propagation. Second, it helps ensure that destructors always accomplish everything they are supposed to accomplish. Each argument is convincing in its own right, but together, the case is ironclad. Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function. Throwing Exceptions Compared to Calling Functions 63 << is the syntax for passing a parameter to a function and throwing an exception... Let me consider a function: class Widget { ... }; // some class; it really // doesn't matter what it is void f1(Widget w); // all these functions void f2(Widget& w); // take parameters of void f3(const Widget& w); // type Widget, Widget&, or void f4(Widget *pw); // Widget* void f5(const Widget *pw); catch (Widget w) ... // all these catch clauses catch (Widget& w) ... // catch exceptions of catch (const Widget& w) ... // type Widget, Widget&, or catch (Widget *pw) ... // Widget* catch (const Widget *pw) ... You might therefore assume that passing an exception from a throw site to a catch clause is basically the same as passing an argument from a function call site to the function's parameter. There are some similarities, to be sure, but there are significant differences, too. Let us begin with a similarity. You can pass both function parameters and exceptions by value, by reference, or by pointer. What happens when you pass parameters and exceptions, however, is quite different. This difference grows out of the fact that when you call a function, control eventually returns to the site of the call (unless the function fails to return), but when you throw an exception, control does not return to the throw site. Consider a function that both passes a Widget as a parameter and throws a Widget as an exception: // function to read the value of a Widget from a stream istream operator>>(istream& s, Widget& w); void passAndThrowWidget() { Widget localWidget; cin >> localWidget; // pass localWidget to operator>> throw localWidget; // throw localWidget as an exception } When localWidget is passed to operator>>, no copying is performed. Instead, the reference w inside operator>> is bound to localWidget, and anything done to w is really done to localWidget. It's a different story when localWidget is thrown as an exception. Regardless of whether the exception is caught by value or by reference (it can't be caught by pointer — that would be a type mismatch), a copy of localWidget will be made, and it is the copy that is passed to the catch clause. This must be the case, because localWidget will go out of scope once control leaves passAndThrowWidget, and when localWidget goes out of scope, its destructor will be called. If localWidget itself were passed to a catch clause, the clause would receive a destructed Widget, an ex-Widget, a former Widget, the carcass of what once was but is no longer a Widget. That would not be useful, and that's why C++ specifies that an object thrown as an exception is copied. This copying occurs even if the object being thrown is not in danger of being destroyed. For example, if passAndThrowWidget declares localWidget to be static, void passAndThrowWidget() { static Widget localWidget; // this is now static; it // will exist until the // end of the program cin >> localWidget; // this works as before throw localWidget; // a copy of localWidget is } // still made and thrown a copy of localWidget would still be made when the exception was thrown. This means that even if the exception is caught by reference, it is not possible for the catch block to modify localWidget; it can only modify a copy of localWidget. This mandatory copying of exception objects helps explain another difference between parameter passing and throwing an exception: the latter is typically much slower than the former (see Item 15). When an object is copied for use as an exception, the copying is performed by the object's copy constructor. This copy constructor is the one in the class corresponding to the object's static type, not its dynamic type. For example, consider this slightly modified version of passAndThrowWidget: class Widget { ... }; class SpecialWidget: public Widget { ... }; void passAndThrowWidget() { SpecialWidget localSpecialWidget; ... Widget& rw = localSpecialWidget; // rw refers to a // SpecialWidget throw rw; // this throws an // exception of type // Widget! } Here a Widget exception is thrown, even though rw refers to a SpecialWidget. That's because rw's static type is Widget, not SpecialWidget. That rw actually refers to a SpecialWidget is of no concern to your compilers; all they care about is rw's static type. This behavior may not be what you want, but it's consistent with all other cases in which C++ copies objects. Copying is always based on an object's static type (but see Item 25 for a technique that lets you make copies on the basis of an object's dynamic type). The fact that exceptions are copies of other objects has an impact on how you propagate exceptions from a catch block. Consider these two catch blocks, which at first glance appear to do the same thing: † Compiler writers are actually allowed a slight bit of leeway regarding the "mandatory" nature of the copying; it can be eliminated under certain circumstances. Similar leeway provides the foundation for the return value optimization (see Item 20). 64 Item 12 catch (Widget& w) // catch Widget exceptions { ... // handle the exception throw; // rethrow the exception so it } // continues to propagate catch (Widget& w) // catch Widget exceptions { ... // handle the exception throw w; // propagate a copy of the } // caught exception The only difference between these blocks is that the first one rethrows the current exception, while the second one throws a new copy of the current exception. Setting aside the performance cost of the additional copy operation, is there a difference between these approaches? There is. The first block rethrows the current exception, regardless of its type. In particular, if the exception originally thrown was of type SpecialWidget, the first block would propagate a SpecialWidget exception, even though w's static type is Widget. This is because no copy is made when the exception is rethrown. The second catch block throws a new exception, which will always be of type Widget, because that's w's static type. In general, you'll want to use the throw; syntax to rethrow the current exception, because there's no chance that that will change the type of the exception being propagated. Furthermore, it's more efficient, because there's no need to generate a new exception object. (Incidentally, the copy made for an exception is a temporary object. As Item 19 explains, this gives compilers the right to optimize it out of existence. I wouldn't expect your compilers to work that hard, however. Exceptions are supposed to be rare, so it makes little sense for compiler vendors to pour a lot of energy into their optimization.) Let us examine the three kinds of catch clauses that could catch the Widget exception thrown by passAndThrowWidget. They are: catch (Widget w) ... // catch exception by value catch (Widget& w) ... // catch exception by // reference catch (const Widget& w) ... // catch exception by // reference-to-const Right away we notice another difference between parameter passing and exception propagation. A thrown object (which, as explained above, is always a temporary) may be caught by simple reference; it need not be caught by reference-to-const. Passing a temporary object to a non-const reference parameter is not allowed for function calls (see Item 19), but it is for exceptions. Let us overlook this difference, however, and return to our examination of copying exception objects. We know that when we pass a function argument by value, we make a copy of the passed object, and we store that copy in a function parameter. The same thing happens when we pass an exception by value. Thus, when we declare a catch clause like this, catch (Widget w) ... // catch by value we expect to pay for the creation of two copies of the thrown object, one to create the temporary that all exceptions generate, the second to copy that temporary into w. Similarly, when we catch an exception by reference, catch (Widget& w) ... // catch by reference catch (const Widget& w) ... // also catch by reference we still expect to pay for the creation of a copy of the exception: the copy that is the temporary. In contrast, when we pass function parameters by reference, no copying takes place. When throwing an exception, then, we expect to construct (and later destruct) one more copy of the thrown object than if we passed the same object to a function. We have not yet discussed throwing exceptions by pointer, but throw by pointer is equivalent to pass by pointer. Either way, a copy of the pointer is passed. About all you need to remember is not to throw a pointer to a local object, because that local object will be destroyed when the exception leaves the local object's scope. The catch clause would then be initialized with a pointer to an object that had already been destroyed. This is the behavior the mandatory copying rule is designed to avoid. The way in which objects are moved from call or throw sites to parameters or catch clauses is one way in which argument passing differs from exception propagation. A second difference lies in what constitutes a type match between caller or thrower and callee or catcher. Consider the sqrt function from the standard math library: double sqrt(double); // from <cmath> or <math.h> We can determine the square root of an integer like this: int i; double sqrtOfi = sqrt(i); 66 Item 12 There is nothing surprising here. The language allows implicit conversion from int to double, so in the call to sqrt, i is silently converted to a double, and the result of sqrt corresponds to that double. (See Item 5 for a fuller discussion of implicit type conversions.) In general, such conversions are not applied when matching exceptions to catch clauses. In this code, void f(int value) { try { if (someFunction()) { // if someFunction() returns throw value; // true, throw an int ... } } catch (double d) { // handle exceptions of ... // type double here } ... } the int exception thrown inside the try block will never be caught by the catch clause that takes a double. That clause catches only exceptions that are exactly of type double; no type conversions are applied. As a result, if the int exception is to be caught, it will have to be by some other (dynamically enclosing) catch clause taking an int or an int& (possibly modified by const or volatile). Two kinds of conversions are applied when matching exceptions to catch clauses. The first is inheritance-based conversions. A catch clause for base class exceptions is allowed to handle exceptions of (publicly) derived class types, too. For example, consider the diagnostics portion of the hierarchy of exceptions defined by the standard C++ library: exception logic_error domain_error runtime_error length_error invalid_argument range_error out_of_range overflow_error underflow_error Throwing Exceptions Compared to Calling Functions 67 A catch clause for runtime_errors can catch exceptions of type range_error, overflow_error, and underflow_error, too, and a catch clause accepting an object of the root class exception can catch any kind of exception derived from this hierarchy. This inheritance-based exception-conversion rule applies to values, references, and pointers in the usual fashion (though Item 13 explains why catching values or pointers is generally a bad idea): catch (runtime_error) ... // can catch errors of type catch (runtime_error&) ... // runtime_error, catch (const runtime_error&) ... // range_error, or // overflow_error catch (runtime_error*) ... // can catch errors of type catch (const runtime_error*) ... // runtime_error*, // range_error*, or // overflow_error* The second type of allowed conversion is from a typed to an untyped pointer, so a catch clause taking a const void* pointer will catch an exception of any pointer type: catch (const void*) ... // catches any exception // that's a pointer The final difference between passing a parameter and propagating an exception is that catch clauses are always tried in the order of their appearance. Hence, it is possible for an exception of a (publicly) derived class type to be handled by a catch clause for one of its base class types — even when a catch clause for the derived class is associated with the same try block! For example, try { ... } catch (logic_error& ex) { // this block will catch ... // all logic_error } // exceptions, even those // of derived types catch (invalid_argument& ex) { // this block can never be ... // executed, because all } // invalid_argument // exceptions will be caught // by the clause above Contrast this behavior with what happens when you call a virtual function. When you call a virtual function, the function invoked is the one in the class closest to the dynamic type of the object invoking the function. You might say that virtual functions employ a "best fit" algorithm, while exception handling follows a "first fit" strategy. Compilers may warn you if a catch clause for a derived class comes after one for a base class (some issue an error, because such code used to be illegal 68 Item 13 in C++), but your best course of action is preemptive: never put a catch clause for a base class before a catch clause for a derived class. The code above, for example, should be reordered like this: try { ... } catch (invalid_argument& ex) { // handle invalid_argument ... // exceptions here } catch (logic_error& ex) { // handle all other ... // logic_errors here } There are thus three primary ways in which passing an object to a function or using that object to invoke a virtual function differs from throwing the object as an exception. First, exception objects are always copied; when caught by value, they are copied twice. Objects passed to function parameters need not be copied at all. Second, objects thrown as exceptions are subject to fewer forms of type conversion than are objects passed to functions. Finally, catch clauses are examined in the order in which they appear in the source code, and the first one that can succeed is selected for execution. When an object is used to invoke a virtual function, the function selected is the one that provides the best match for the type of the object, even if it's not the first one listed in the source code. Item 13: Catch exceptions by reference. Catching Exceptions by Reference When you write a catch clause, you must specify how exception objects are to be passed to that clause. You have three choices, just as when specifying how parameters should be passed to functions: by pointer, by value, or by reference. Catching Exceptions by Reference Let us consider first catch by pointer. In theory, this should be the least inefficient way to implement the invariably slow process of moving an exception from throw site to catch clause (see Item 15). That's because throw by pointer is the only way of moving exception information without copying an object (see Item 12). For example: class exception { ... }; // from the standard C++ // library exception // hierarchy (see Item 12) void someFunction() { static exception ex; // exception object ... throw &ex; // throw a pointer to ex ... } void doSomething() { try { someFunction(); // may throw an exception* } catch (exception *ex) { // catches the exception*; ... // no object is copied } } This looks neat and tidy, but it's not quite as well-kept as it appears. For this to work, programmers must define exception objects in a way that guarantees the objects exist after control leaves the functions throwing pointers to them. Global and static objects work fine, but it's easy for programmers to forget the constraint. If they do, they typically end up writing code like this: void someFunction() { exception ex; // local exception object; // will be destroyed when // this function's scope is // exited ... throw &ex; // throw a pointer to an ... // object that's about to } // be destroyed This is worse than useless, because the catch clause handling this exception receives a pointer to an object that no longer exists. An alternative is to throw a pointer to a new heap object: void someFunction() { ... throw new exception; // throw a pointer to a new heap- ... // based object (and hope that // operator new — see Item 8 — // doesn't itself throw an // exception!) } This avoids the I-just-caught-a-pointer-to-a-destroyed-object problem, but now authors of catch clauses confront a nasty question: should they delete the pointer they receive? If the exception object was allocated on the heap, they must, otherwise they suffer a resource leak. If the exception object wasn't allocated on the heap, they mustn't, otherwise they suffer undefined program behavior. What to do? It's impossible to know. Some clients might pass the address of a global or static object, others might pass the address of an exception on the heap. Catch by pointer thus gives rise to the Hamlet conundrum: to delete or not to delete? It's a question with no good answer. You're best off ducking it. Furthermore, catch-by-pointer runs contrary to the convention established by the language itself. The four standard exceptions — bad_alloc (thrown when operator new (see Item 8) can't satisfy a memory request), bad_cast (thrown when a dynamic_cast to a reference fails; see Item 2), bad_typeid (thrown when typeid is applied to a dereferenced null pointer), and bad_exception (available for unexpected exceptions; see Item 14) — are all objects, not pointers to objects, so you have to catch them by value or by reference, anyway. Catch-by-value eliminates questions about exception deletion and works with the standard exception types. However, it requires that exception objects be copied twice each time they're thrown (see Item 12). It also gives rise to the specter of the slicing problem, whereby derived class exception objects caught as base class exceptions have their derivedness "sliced" off. Such "sliced" objects are base class objects: they lack derived class data members, and when virtual functions are called on them, they resolve to virtual functions of the base class. (Exactly the same thing happens when an object is passed to a function by value.) For example, consider an application employing an exception class hierarchy that extends the standard one: class exception { // as above, this is a public: // standard exception class virtual const char * what() throw(); // returns a brief descrip. ... // of the exception (see // Item 14 for info about }; // the "throw()" at the // end of the declaration) class runtime_error: // also from the standard public exception { ... }; // C++ exception hierarchy class Validation_error: // this is a class added by public runtime_error { // a client public: virtual const char * what() throw(); // this is a redefinition ... // of the function declared }; // in class exception above void someFunction() // may throw a validation { // exception ... if (a validation test fails) { throw Validation_error(); } ... } 70 Item 13 void doSomething() { try { someFunction(); // may throw a validation } // exception catch (exception ex) { // catches all exceptions // in or derived from // the standard hierarchy cerr << ex.what(); // calls exception::what(), ... // never // Validation_error::what() } } The version of what that is called is that of the base class, even though the thrown exception is of type Validation_error and Validation_error redefines that virtual function. This kind of slicing behavior is almost never what you want. That leaves only catch-by-reference. Catch-by-reference suffers from none of the problems we have discussed. Unlike catch-by-pointer, the question of object deletion fails to arise, and there is no difficulty in catching the standard exception types. Unlike catch-by-value, there is no slicing problem, and exception objects are copied only once. If we rewrite the last example using catch-by-reference, it looks like this: void someFunction() // nothing changes in this { // function ... if (a validation test fails) { throw Validation_error(); } ... } void doSomething() { try { someFunction(); // no change here } catch (exception& ex) { // here we catch by reference // instead of by value cerr << ex.what(); // now calls // Validation_error::what(), ... // not exception::what() } } There is no change at the throw site, and the only change in the catch clause is the addition of an ampersand. This tiny modification makes a big difference, however, because virtual functions in the catch block now work as we expect: functions in Validation_error are invoked if they redefine those in exception. Of course, if there is no need to modify the exception object in the handler, you'd catch not just by reference, but by reference to const. What a happy confluence of felicities! If you catch by reference, you sidestep questions about object deletion that leave you damned if you do and damned if you don't; you avoid slicing exception objects; you retain the ability to catch standard exceptions; and you limit the number of times exception objects need to be copied. So what are you waiting for? Catch exceptions by reference! ... 71 Item 14 may throw a Validation_error exception. It is not enough to say... char *p = &s1[1]; Our data structure at this point looks like this: s1 1 Hello p Now consider an additional statement: String s2 = s1; 192 Item 29 The String copy constructor will make s2 share s1's StringValue, so the resulting data structure will be this one: s1 2 Hello s2 p The implications of a statement such as the following, then, are not pleasant to contemplate: *p = 'x'; // modifies both s1 and s2! There is no way the String copy constructor can detect this problem, because it has no way to know that a pointer into s1's StringValue object exists. And this problem isn't limited to pointers: it would exist if someone had saved a reference to the result of a call to String's nonconst operator[]. There are at least three ways of dealing with this problem. The first is to ignore it, to pretend it doesn't exist. This approach turns out to be distressingly common in class libraries that implement reference-counted strings. If you have access to a reference-counted string, try the above example and see if you're distressed, too. If you're not sure if you have access to a reference-counted string, try the example anyway. Through the wonder of encapsulation, you may be using such a type without knowing it. Not all implementations ignore such problems. A slightly more sophisticated way of dealing with such difficulties is to define them out of existence. Implementations adopting this strategy typically put something in their documentation that says, more or less, "Don't do that. If you do, results are undefined." If you then do it anyway — wittingly or no — and complain about the results, they respond, "Well, we told you not to do that." Such implementations are often efficient, but they leave much to be desired in the usability department. There is a third solution, and that's to eliminate the problem. It's not difficult to implement, but it can reduce the amount of value sharing between objects. Its essence is this: add a flag to each StringValue object indicating whether that object is shareable. Turn the flag on initially (the object is shareable), but turn it off whenever the nonconst operator[] is invoked on the value represented by that object. Once the flag is set to false, it stays that way forever.† The string type in the standard C++ library (see Item 35) uses a combination of solutions two and three. The reference returned from the non-const operator[] is guaranteed to be valid until the next function call that might modify the string. After that, use of the reference (or the character to which it refers) yields undefined results. This allows the string's shareability flag to be reset to true whenever a function is called that might modify the string. Reference Counting 193 Here's a modified version of StringValue that includes a shareability flag: class String { private: struct StringValue { int refCount; bool shareable; // add this char *data; StringValue(const char *initValue); ~StringValue(); }; ... }; String::StringValue::StringValue(const char *initValue) : refCount(1), shareable(true) // add this { data = new char[strlen(initValue) + 1]; strcpy(data, initValue); } String::StringValue::~StringValue() { delete [] data; } As you can see, not much needs to change; the two lines that require modification are flagged with comments. Of course, String's member functions must be updated to take the shareable field into account. Here's how the copy constructor would do that: String::String(const String& rhs) { if (rhs.value->shareable) { value = rhs.value; ++value->refCount; } else { value = new StringValue(rhs.value->data); } } All the other String member functions would have to check the shareable field in an analogous fashion. The non-const version of operator[] would be the only function to set the shareable flag to false: 194 Item 29 char& String::operator[](int index) { if (value->refCount > 1) { --value->refCount; value = new StringValue(value->data); } value->shareable = false; // add this return value->data[index]; } If you use the proxy class technique of Item 30 to distinguish read usage from write usage in operator[], you can usually reduce the number of StringValue objects that must be marked unshareable. A Reference-Counting Base Class Reference counting is useful for more than just strings. Any class in which different objects may have values in common is a legitimate candidate for reference counting. Rewriting a class to take advantage of reference counting can be a lot of work, however, and most of us already have more than enough to do. Wouldn't it be nice if we could somehow write (and test and document) the reference counting code in a context-independent manner, then just graft it onto classes when needed? Of course it would. In a curious twist of fate, there's a way to do it (or at least to do most of it). The first step is to create a base class, RCObject, for reference-counted objects. Any class wishing to take advantage of automatic reference counting must inherit from this class. RCObject encapsulates the reference count itself, as well as functions for incrementing and decrementing that count. It also contains the code for destroying a value when it is no longer in use, i.e., when its reference count becomes 0. Finally, it contains a field that keeps track of whether this value is shareable, and it provides functions to query this value and set it to false. There is no need for a function to set the shareability field to true, because all values are shareable by default. As noted above, once an object has been tagged unshareable, there is no way to make it shareable again. RCObject's class definition looks like this: class RCObject { public: RCObject(); RCObject(const RCObject& rhs); RCObject& operator=(const RCObject& rhs); virtual ~RCObject() = 0; void addReference(); void removeReference(); void markUnshareable(); bool isShareable() const; bool isShared() const; private: int refCount; bool shareable; }; RCObjects can be created (as the base class parts of more derived objects) and destroyed; they can have new references added to them and can have current references removed; their shareability status can be queried and can be disabled; and they can report whether they are currently being shared. That's all they offer. As a class encapsulating the notion of being reference-countable, that's really all we have a right to expect them to do. Note the tell-tale virtual destructor, a sure sign this class is designed for use as a base class. Note also how the destructor is a pure virtual function, a sure sign this class is designed to be used only as a base class. The code to implement RCObject is, if nothing else, brief: RCObject::RCObject() : refCount(0), shareable(true) {} RCObject::RCObject(const RCObject&) : refCount(0), shareable(true) {} RCObject& RCObject::operator=(const RCObject&) { return *this; } RCObject::~RCObject() {} // virtual dtors must always // be implemented, even if // they are pure virtual // and do nothing (see also // Item 33) void RCObject::addReference() { ++refCount; } void RCObject::removeReference() { if (--refCount == 0) delete this; } void RCObject::markUnshareable() { shareable = false; } bool RCObject::isShareable() const { return shareable; } bool RCObject::isShared() const { return refCount > 1; } 196 Item 29 Curiously, we set refCount to 0 inside both constructors. This seems counterintuitive. Surely at least the creator of the new RCObject is referring to it! As it turns out, it simplifies things for the creators of RCObjects to set refCount to 1 themselves, so we oblige them here by not getting in their way. We'll get a chance to see the resulting code simplification shortly. Another curious thing is that the copy constructor always sets refCount to 0, regardless of the value of refCount for the RCObject we're copying. That's because we're creating a new object representing a value, and new values are always unshared and referenced only by their creator. Again, the creator is responsible for setting the refCount to its proper value. The RCObject assignment operator looks downright subversive: it does nothing. Frankly, it's unlikely this operator will ever be called. RCObject is a base class for a shared value object, and in a system based on reference counting, such objects are not assigned to one another, objects pointing to them are. In our case, we don't expect StringValue objects to be assigned to one another, we expect only String objects to be involved in assignments. In such assignments, no change is made to the value of a StringValue — only the StringValue reference count is modified. Nevertheless, it is conceivable that some as-yet-unwritten class might someday inherit from RCObject and might wish to allow assignment of reference-counted values (see Item 32). If so, RCObject's assignment operator should do the right thing, and the right thing is to do nothing. To see why, imagine that we wished to allow assignments between StringValue objects. Given StringValue objects sv1 and sv2, what should happen to sv1's and sv2's reference counts in an assignment? sv1 = sv2; // how are sv1's and sv2's reference // counts affected? Before the assignment, some number of String objects are pointing to sv1. That number is unchanged by the assignment, because only sv1's value changes. Similarly, some number of String objects are pointing to sv2 prior to the assignment, and after the assignment, exactly the same String objects point to sv2. sv2's reference count is also unchanged. When RCObjects are involved in an assignment, then, the number of objects pointing to those objects is unaffected, hence RCObject::operator= should change no reference counts. That's exactly what the implementation above does. Counterintuitive? Perhaps, but it's still correct. The code for RCObject::removeReference is responsible not only for decrementing the object's refCount, but also for destroying the object if the new value of refCount is 0. It accomplishes this latter task by deleting this, which, as Item 27 explains, is safe only if we know that *this is a heap object. For this class to be successful, we must engineer things so that RCObjects can be created only on the heap. General approaches to achieving that end are discussed in Item 27, but the specific measures we'll employ in this case are described at the conclusion of this Item. To take advantage of our new reference-counting base class, we modify StringValue to inherit its reference counting capabilities from RCObject: class String { private: struct StringValue: public RCObject { char *data; StringValue(const char *initValue); ~StringValue(); }; ... }; String::StringValue::StringValue(const char *initValue) { data = new char[strlen(initValue) + 1]; strcpy(data, initValue); } String::StringValue::~StringValue() { delete [] data; } This version of StringValue is almost identical to the one we saw earlier. The only thing that's changed is that StringValue's member functions no longer manipulate the refCount field. RCObject now handles what they used to do. Don't feel bad if you blanched at the sight of a nested class (StringValue) inheriting from a class (RCObject) that's unrelated to the nesting class (String). It looks weird to everybody at first, but it's perfectly kosher. A nested class is just as much a class as any other, so it has the freedom to inherit from whatever other classes it likes. In time, you won't think twice about such inheritance relationships. 198 Item 29 Automating Reference Count Manipulations The RCObject class gives us a place to store a reference count, and it gives us member functions through which that reference count can be manipulated, but the calls to those functions must still be manually inserted in other classes. It is still up to the String copy constructor and the String assignment operator to call addReference and removeReference on StringValue objects. This is clumsy. We'd like to move those calls out into a reusable class, too, thus freeing authors of classes like String from worrying about any of the details of reference counting. Can it be done? Isn't C++ supposed to support reuse? It can, and it does. There's no easy way to arrange things so that all reference-counting considerations can be moved out of application classes, but there is a way to eliminate most of them for most classes. (In some application classes, you can eliminate all reference-counting code, but our String class, alas, isn't one of them. One member function spoils the party, and I suspect you won't be too surprised to hear it's our old nemesis, the non-const version of operator[]. Take heart, however; we'll tame that miscreant in the end.) Notice that each String object contains a pointer to the StringValue object representing that String's value: class String { private: struct StringValue: public RCObject { ... }; StringValue *value; // value of this String ... }; We have to manipulate the refCount field of the StringValue object anytime anything interesting happens to one of the pointers pointing to it. "Interesting happenings" include copying a pointer, reassigning one, and destroying one. If we could somehow make the pointer itself detect these happenings and automatically perform the necessary manipulations of the refCount field, we'd be home free. Unfortunately, pointers are rather dense creatures, and the chances of them detecting anything, much less automatically reacting to things they detect, are pretty slim. Fortunately, there's a way to smarten them up: replace them with objects that act like pointers, but that do more. Such objects are called smart pointers, and you can read about them in more detail than you probably care to in Item 28. For our purposes here, it's enough to know that smart pointer objects support the member selection (->) and dereferencing (*) operations, just like real pointers (which, in this context, are generally referred to as dumb pointers). Reference Counting 199 and, like dumb pointers, they are strongly typed: you can't make a smart pointer-to-T point to an object that isn't of type T. Here's a template for objects that act as smart pointers to reference-counted objects: // template class for smart pointers-to-T objects. T must // support the RCObject interface, typically by inheriting // from RCObject template<class T> class RCPtr { public: RCPtr(T* realPtr = 0); RCPtr(const RCPtr& rhs); ~RCPtr(); RCPtr& operator=(const RCPtr& rhs); T* operator->() const; // see Item 28 T& operator*() const; // see Item 28 private: T *pointee; // dumb pointer this // object is emulating void init(); // common initialization }; // code This template gives smart pointer objects control over what happens during their construction, assignment, and destruction. When such events occur, these objects can automatically perform the appropriate manipulations of the refCount field in the objects to which they point. For example, when an RCPtr is created, the object it points to needs to have its reference count increased. There's no need to burden application developers with the requirement to tend to this irksome detail manually, because RCPtr constructors can handle it themselves. The code in the two constructors is all but identical — only the member initialization lists differ — so rather than write it twice, we put it in a private member function called init and have both constructors call that: template<class T> RCPtr<T>::RCPtr(T* realPtr): pointee(realPtr) { init(); } template<class T> RCPtr<T>::RCPtr(const RCPtr& rhs): pointee(rhs.pointee) { init(); } 200 Item 29 template<class T> void RCPtr<T>::init() { if (pointee == 0) { // if the dumb pointer is return; // null, so is the smart one } if (pointee->isShareable() == false) { // if the value pointee = new T(*pointee); // isn't shareable, } // copy it pointee->addReference(); // note that there is now a } // new reference to the value Moving common code into a separate function like init is exemplary software engineering, but its luster dims when, as in this case, the function doesn't behave correctly. The problem is this. When init needs to create a new copy of a value (because the existing copy isn't shareable), it executes the following code: pointee = new T(*pointee); The type of pointee is pointer-to-T, so this statement creates a new T object and initializes it by calling T's copy constructor. In the case of an RCPtr in the String class, T will be String::StringValue, so the statement above will call String::StringValue's copy constructor. We haven't declared a copy constructor for that class, however, so our compilers will generate one for us. The copy constructor so generated will, in accordance with the rules for automatically generated copy constructors in C++, copy only StringValue's data pointer; it will not copy the char* string data points to. Such behavior is disastrous in nearly any class (not just reference-counted classes), and that's why you should get into the habit of writing a copy constructor (and an assignment operator) for all your classes that contain pointers (see Item E11). Reference Counting 201 The correct behavior of the RCPtr<T> template depends on T containing a copy constructor that makes a truly independent copy (i.e., a deep copy) of the value represented by T. We must augment StringValue with such a constructor before we can use it with the RCPtr class: class String { private: struct StringValue: public RCObject { StringValue(const StringValue& rhs); ... }; ... }; String::StringValue::StringValue(const StringValue& rhs) { data = new char[strlen(rhs.data) + 1]; strcpy(data, rhs.data); } The existence of a deep-copying copy constructor is not the only assumption RCPtr<T> makes about T. It also requires that T inherit from RCObject, or at least that T provide all the functionality that RCObject does. In view of the fact that RCPtr objects are designed to point only to reference-counted objects, this is hardly an unreasonable assumption. Nevertheless, the assumption must be documented. A final assumption in RCPtr<T> is that the type of the object pointed to is T. This seems obvious enough. After all, pointee is declared to be of type T*. But pointee might really point to a class derived from T. For example, if we had a class SpecialStringValue that inherited from String::StringValue, class String { private: struct StringValue: public RCObject { ... }; struct SpecialStringValue: public StringValue { ... }; ... }; 202 Item 29 we could end up with a String containing a RCPtr<StringValue> pointing to a SpecialStringValue object. In that case, we'd want this part of init, pointee = new T(*pointee); // T is StringValue, but // pointee really points to // a SpecialStringValue to call SpecialStringValue's copy constructor, not StringValue's. We can arrange for this to happen by using a virtual copy constructor (see Item 25). In the case of our String class, we don't expect classes to derive from StringValue, so we'll disregard this issue. With RCPtr's constructors out of the way, the rest of the class's functions can be dispatched with considerably greater alacrity. Assignment of an RCPtr is straightforward, though the need to test whether the newly assigned value is shareable complicates matters slightly. Fortunately, such complications have already been handled by the init function that was created for RCPtr's constructors. We take advantage of that fact by using it again here: template<class T> RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs) { if (pointee != rhs.pointee) { // skip assignments // where the value // doesn't change T *oldPointee = pointee; // save old pointee value pointee = rhs.pointee; // point to new value init(); // if possible, share it // else make own copy if (oldPointee) { oldPointee->removeReference(); // remove reference to } // current value } return *this; } The destructor is easier. When an RCPtr is destroyed, it simply removes its reference to the reference-counted object: template<class T> RCPtr<T>::~RCPtr() { if (pointee)pointee->removeReference(); } Reference Counting 203 If the RCPtr that just expired was the last reference to the object, that object will be destroyed inside RCObject's removeReference member function. Hence RCPtr objects never need to worry about destroying the values they point to. Finally, RCPtr's pointer-emulating operators are part of the smart pointer boilerplate you can read about in Item 28: template<class T> T* RCPtr<T>::operator->() const { return pointee; } template<class T> T& RCPtr<T>::operator*() const { return *pointee; } Putting it All Together Enough! Finis! At long last we are in a position to put all the pieces together and build a reference-counted String class based on the reusable RCObject and RCPtr classes. With luck, you haven't forgotten that that was our original goal. Each reference-counted string is implemented via this data structure: RCObject class public inheritance String object StringValue object pointer RCPtr object Heap Memory The classes making up this data structure are defined like this: template<class T> // template class for smart class RCPtr { // pointers-to-T objects; T public: // must inherit from RCObject RCPtr(T* realPtr = 0); RCPtr(const RCPtr& rhs); ~RCPtr(); RCPtr& operator=(const RCPtr& rhs); T* operator->() const; T& operator*() const; private: T *pointee; void init(); }; 204 Item 29 class RCObject { // base class for reference- public: // counted objects void addReference(); void removeReference(); void markUnshareable(); bool isShareable() const; bool isShared() const; protected: RCObject(); RCObject(const RCObject& rhs); RCObject& operator=(const RCObject& rhs); virtual ~RCObject() = 0; private: int refCount; bool shareable; }; class String { // class to be used by public: // application developers String(const char *value = ""); const char& operator[](int index) const; char& operator[](int index); private: // class representing string values struct StringValue: public RCObject { char *data; StringValue(const char *initValue); StringValue(const StringValue& rhs); void init(const char *initValue); ~StringValue(); }; RCPtr<StringValue> value; }; For the most part, this is just a recap of what we've already developed, so nothing should be much of a surprise. Close examination reveals we've added an init function to String::StringValue, but, as we'll see below, that serves the same purpose as the corresponding function in RCPtr: it prevents code duplication in the constructors. There is a significant difference between the public interface of this String class and the one we used at the beginning of this Item. Where is the copy constructor? Where is the assignment operator? Where is the destructor? Something is definitely amiss here. Reference Counting 205 Actually, no. Nothing is amiss. In fact, some things are working perfectly. If you don't see what they are, prepare yourself for a C++ epiphany. We don't need those functions anymore. Sure, copying of String objects is still supported, and yes, the copying will correctly handle the underlying reference-counted StringValue objects, but the String class doesn't have to provide a single line of code to make this happen. That's because the compiler-generated copy constructor for String will automatically call the copy constructor for String's RCPtr member, and the copy constructor for that class will perform all the necessary manipulations of the StringValue object, including its reference count. An RCPtr is a smart pointer, remember? We designed it to take care of the details of reference counting, so that's what it does. It also handles assignment and destruction, and that's why String doesn't need to write those functions, either. Our original goal was to move the unreusable reference-counting code out of our hand-written String class and into context-independent classes where it would be available for use with any class. Now we've done it (in the form of the RCObject and RCPtr classes), so don't be so surprised when it suddenly starts working. It's supposed to work. Just so you have everything in one place, here's the implementation of RCObject: RCObject::RCObject() : refCount(0), shareable(true) {} RCObject::RCObject(const RCObject&) : refCount(0), shareable(true) {} RCObject& RCObject::operator=(const RCObject&) { return *this; } RCObject::~RCObject() {} void RCObject::addReference() { ++refCount; } void RCObject::removeReference() { if (--refCount == 0) delete this; } void RCObject::markUnshareable() { shareable = false; } bool RCObject::isShareable() const { return shareable; } bool RCObject::isShared() const { return refCount > 1; } And here's the implementation of RCPtr: 206 Item 29 template<class T> void RCPtr<T>::init() { if (pointee == 0) return; if (pointee->isShareable() == false) { pointee = new T(*pointee); } pointee->addReference(); } template<class T> RCPtr<T>::RCPtr(T* realPtr) : pointee(realPtr) { init(); } template<class T> RCPtr<T>::RCPtr(const RCPtr& rhs) : pointee(rhs.pointee) { init(); } template<class T> RCPtr<T>::~RCPtr() { if (pointee) pointee->removeReference(); } template<class T> RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs) { if (pointee != rhs.pointee) { T *oldPointee = pointee; pointee = rhs.pointee; init(); if (oldPointee) oldPointee->removeReference(); } return *this; } template<class T> T* RCPtr<T>::operator->() const { return pointee; } template<class T> T& RCPtr<T>::operator*() const { return *pointee; } Reference Counting 207 The implementation of String::StringValue looks like this: void String::StringValue::init(const char *initValue) { data = new char[strlen(initValue) + 1]; strcpy(data, initValue); } String::StringValue::StringValue(const char *initValue) { init(initValue); } String::StringValue::StringValue(const StringValue& rhs) { init(rhs.data); } String::StringValue::~StringValue() { delete [] data; } Ultimately, all roads lead to String, and that class is implemented this way: String::String(const char *initValue) : value(new StringValue(initValue)) {} const char& String::operator[](int index) const { return value->data[index]; } char& String::operator[](int index) { if (value->isShared()) { value = new StringValue(value->data); } value->markUnshareable(); return value->data[index]; } If you compare the code for this String class with that we developed for the String class using dumb pointers, you'll be struck by two things. First, there's a lot less of it here than there. That's because RCPtr has assumed much of the reference-counting burden that used to fall on String. Second, the code that remains in String is nearly unchanged: the smart pointer replaced the dumb pointer essentially seamlessly. In fact, the only changes are in operator[], where we call isShared instead of checking the value of refCount directly and where our use of the smart RCPtr object eliminates the need to manually manipulate the reference count during a copy-on-write. This is all very nice, of course. Who can object to less code? Who can oppose encapsulation success stories? The bottom line, however, is determined more by the impact of this newfangled String class on its clients than by any of its implementation details, and it is here that things really shine. If no news is good news, the news here is very good indeed. The String interface has not changed. We added reference counting, we added the ability to mark individual string values as unshareable, we moved the notion of reference countability into a new base class, we added smart pointers to automate the manipulation of reference counts, yet not one line of client code needs to be changed. Sure, we changed the String class definition, so clients who want to take advantage of reference-counted strings must recompile and relink, but their investment in code is completely and utterly preserved. You see? Encapsulation really is a wonderful thing. 208 Item 29 Adding Reference Counting to Existing Classes Everything we've discussed so far assumes we have access to the source code of the classes we're interested in. But what if we'd like to apply the benefits of reference counting to some class Widget that's in a library we can't modify? There's no way to make Widget inherit from RCObject, so we can't use smart RCPtrs with it. Are we out of luck? We're not. With some minor modifications to our design, we can add reference counting to any type. First, let's consider what our design would look like if we could have Widget inherit from RCObject. In that case, we'd have to add a class, RCWidget, for clients to use, but everything would then be analogous to our String/StringValue example, with RCWidget playing the role of String and Widget playing the role of StringValue. The design would look like this: RCObject class public inheritance RCWidget object Widget object pointer RCPtr object We can now apply the maxim that most problems in Computer Science can be solved with an additional level of indirection. We add a new class, CountHolder, to hold the reference count, and we have CountHolder inherit from RCObject. We also have CountHolder contain a pointer to a Widget. We then replace the smart RCPtr template with an equally smart RCIPtr template that knows about the existence of the CountHolder class. (The "I" in RCIPtr stands for "indirect.") The modified design looks like this: RCObject class public inheritance RCWidget object CountHolder object pointer RCIPtr object pointer Widget object Reference Counting 209 Just as StringValue was an implementation detail hidden from clients of String, CountHolder is an implementation detail hidden from clients of RCWidget. In fact, it's an implementation detail of RCIPtr, so it's nested inside that class. RCIPtr is implemented this way: template<class T> class RCIPtr { public: RCIPtr(T* realPtr = 0); RCIPtr(const RCIPtr& rhs); ~RCIPtr(); RCIPtr& operator=(const RCIPtr& rhs); const T* operator->() const; // see below for an T* operator->(); // explanation of why const T& operator*() const; // these functions are T& operator*(); // declared this way private: struct CountHolder: public RCObject { ~CountHolder() { delete pointee; } T *pointee; }; CountHolder *counter; void init(); void makeCopy(); // see below }; template<class T> void RCIPtr<T>::init() { if (counter->isShareable() == false) { T *oldValue = counter->pointee; counter = new CountHolder; counter->pointee = oldValue ? new T(*oldValue) : 0; } counter->addReference(); } template<class T> RCIPtr<T>::RCIPtr(T* realPtr) : counter(new CountHolder) { counter->pointee = realPtr; init(); } template<class T> RCIPtr<T>::RCIPtr(const RCIPtr& rhs) : counter(rhs.counter) { init(); } template<class T> RCIPtr<T>::~RCIPtr() { counter->removeReference(); } 210 Item 29 template<class T> RCIPtr<T>& RCIPtr<T>::operator=(const RCIPtr& rhs) { if (counter != rhs.counter) { counter->removeReference(); counter = rhs.counter; init(); } return *this; } template<class T> // implement the copy void RCIPtr<T>::makeCopy() // part of copy-on- { // write (COW) if (counter->isShared()) { T *oldValue = counter->pointee; counter->removeReference(); counter = new CountHolder; counter->pointee = oldValue ? new T(*oldValue) : 0; counter->addReference(); } } template<class T> // const access; const T* RCIPtr<T>::operator->() const // no COW needed { return counter->pointee; } template<class T> // non-const T* RCIPtr<T>::operator->() // access; COW { makeCopy(); return counter->pointee; } // needed template<class T> // const access; const T& RCIPtr<T>::operator*() const // no COW needed { return *(counter->pointee); } template<class T> // non-const T& RCIPtr<T>::operator*() // access; do the { makeCopy(); return *(counter->pointee); } // COW thing RCIPtr differs from RCPtr in only two ways. First, RCPtr objects point to values directly, while RCIPtr objects point to values through intervening CountHolder objects. Second, RCIPtr overloads operator-> and operator* to create a copy-on-write when a non-const access is made to a pointed-to object. Given RCIPtr, it's easy to implement RCWidget, because each function in RCWidget is implemented by forwarding the call through the underlying RCIPtr to a Widget object. For example, if Widget looks like this, class Widget { public: Widget(int size); Widget(const Widget& rhs); ~Widget(); Widget& operator=(const Widget& rhs); void doThis(); int showThat() const; }; Reference Counting 211 RCWidget will be defined this way: class RCWidget { public: RCWidget(int size): value(new Widget(size)) {} void doThis() { value->doThis(); } int showThat() const { return value->showThat(); } private: RCIPtr<Widget> value; }; Note how the RCWidget constructor calls the Widget constructor (via the new operator — see Item 8) with the argument it was passed; how RCWidget's doThis calls doThis in the Widget class; and how RCWidget::showThat returns whatever its Widget counterpart returns. Notice also how RCWidget declares no copy constructor, no assignment operator, and no destructor. As with the String class, there is no need to write these functions. Thanks to the behavior of the RCIPtr class, the default versions do the right things. If the thought occurs to you that creation of RCWidget is so mechanical, it could be automated, you're right. It would not be difficult to write a program that takes a class like Widget as input and produces a class like RCWidget as output. If you write such a program, please let me know. Evaluation Let us disentangle ourselves from the details of widgets, strings, values, smart pointers, and reference-counting base classes. That gives us an opportunity to step back and view reference counting in a broader context. In that more general context, we must address a higher-level question, namely, when is reference counting an appropriate technique? Reference-counting implementations are not without cost. Each reference-counted value carries a reference count with it, and most operations require that this reference count be examined or manipulated in some way. Object values therefore require more memory, and we sometimes execute more code when we work with them. Furthermore, the underlying source code is considerably more complex for a reference-counted class than for a less elaborate implementation. An un-reference-counted string class typically stands on its own, while our final String class is useless unless it's augmented with three auxiliary classes (StringValue, RCObject, and RCPtr). True, our more complicated design holds out the promise of greater efficiency when values can be shared, it eliminates the need to track object ownership, and it promotes reusability of the reference counting idea and implementation. Nevertheless, that quartet of classes has to be written, tested, documented, and maintained, and that's going to be more work than writing, testing, documenting, and maintaining a single class. Even a manager can see that. Reference counting is an optimization technique predicated on the assumption that objects will commonly share values (see also Item 18). If this assumption fails to hold, reference counting will use more memory than a more conventional implementation and it will execute more code. On the other hand, if your objects do tend to have common values, 212 Item 29 reference counting should save you both time and space. The bigger your object values and the more objects that can simultaneously share values, the more memory you'll save. The more you copy and assign values between objects, the more time you'll save. The more expensive it is to create and destroy a value, the more time you'll save there, too. In short, reference counting is most useful for improving efficiency under the following conditions: ■ Relatively few values are shared by relatively many objects. Such sharing typically arises through calls to assignment operators and copy constructors. The higher the objects/values ratio, the better the case for reference counting. ■ Object values are expensive to create or destroy, or they use lots of memory. Even when this is the case, reference counting still buys you nothing unless these values can be shared by multiple objects. There is only one sure way to tell whether these conditions are satisfied, and that way is not to guess or rely on your programmer's intuition (see Item 16). The reliable way to find out whether your program can benefit from reference counting is to profile or instrument it. That way you can find out if creating and destroying values is a performance bottleneck, and you can measure the objects/values ratio. Only when you have such data in hand are you in a position to determine whether the benefits of reference counting (of which there are many) outweigh the disadvantages (of which there are also many). Even when the conditions above are satisfied, a design employing reference counting may still be inappropriate. Some data structures (e.g., directed graphs) lead to self-referential or circular dependency structures. Such data structures have a tendency to spawn isolated collections of objects, used by no one, whose reference counts never drop to zero. That's because each object in the unused structure is pointed to by at least one other object in the same structure. Industrial-strength garbage collectors use special techniques to find such structures and eliminate them, but the simple reference-counting approach we've examined here is not easily extended to include such techniques. Reference counting can be attractive even if efficiency is not your primary concern. If you find yourself weighed down with uncertainty over who's allowed to delete what, reference counting could be just the technique you need to ease your burden. Many programmers are devoted to reference counting for this reason alone. Let us close this discussion on a technical note by tying up one remaining loose end. When RCObject::removeReference decrements an object's reference count, it checks to see if the new count is 0. If it is, removeReference destroys the object by deleting this. This is a safe operation only if the object was allocated by calling new, so we need some way of ensuring that RCObjects are only created in that manner. In this case we do it by convention. RCObject is designed for use as a base class of reference-counted value objects, and those value objects should be referred to only by smart RCPtr pointers. Furthermore, the value objects should be instantiated only by application objects that realize values are being shared; the classes describing the value objects should never be available for general use. In our example, the class for value objects is StringValue, and we limit its use by making it private in String. Only String can create StringValue objects, so it is up to the author of the String class to ensure that all such objects are allocated via new. Our approach to the constraint that RCObjects be created only on the heap, then, is to assign responsibility for conformance to this constraint to a well-defined set of classes and to ensure that only that set of classes can create RCObjects. There is no possibility that random clients can accidently (or maliciously) create RCObjects in an inappropriate manner. We limit the right to create reference-counted objects, and when we do hand out that right, we make it clear that it's accompanied by the concomitant responsibility to follow the rules governing object creation. Item 30: Proxy classes. Though your in-laws may be one-dimensional, the world, in general, is not. Unfortunately, C++ hasn't yet caught on to that fact. At least, there's little evidence for it in the language's support for arrays. You can create two-dimensional, three-dimensional — heck, you can create n-dimensional — arrays in FORTRAN, in BASIC, even in COBOL (okay, FORTRAN only allows up to seven dimensions, but let's not quibble), but can you do it in C++? Only sometimes, and even then only sort of. Proxy Classes Proxy Classes This much is legal: int data[10][20]; // 2D array: 10 by 20 The corresponding construct using variables as dimension sizes, however, is not: void processInput(int dim1, int dim2) { int data[dim1][dim2]; // error! array dimensions ... // must be known during } // compilation 214 Item 30 It's not even legal for a heap-based allocation: int *data = new int[dim1][dim2]; // error! Implementing Two-Dimensional Arrays Multidimensional arrays are as useful in C++ as they are in any other language, so it's important to come up with a way to get decent support for them. The usual way is the standard one in C++: create a class to represent the objects we need but that are missing in the language proper. Hence we can define a class template for two-dimensional arrays: template<class T> class Array2D { public: Array2D(int dim1, int dim2); ... }; Now we can define the arrays we want: Array2D<int> data(10, 20); // fine Array2D<float> *data = new Array2D<float>(10, 20); // fine void processInput(int dim1, int dim2) { Array2D<int> data(dim1, dim2); // fine ... } Using these array objects, however, isn't quite as straightforward. In keeping with the grand syntactic tradition of both C and C++, we'd like to be able to use brackets to index into our arrays, cout << data[3][6]; but how do we declare the indexing operator in Array2D to let us do this? Our first impulse might be to declare operator[][] functions, like this: template<class T> class Array2D { public: // declarations that won't compile T& operator[][](int index1, int index2); const T& operator[][](int index1, int index2) const; ... }; We'd quickly learn to rein in such impulses, however, because there is no such thing as operator[][], and don't think your compilers will forget it. (For a complete list of operators, overloadable and otherwise, see Item 7.) We'll have to do something else. If you can stomach the syntax, you might follow the lead of the many programming languages that use parentheses to index into arrays. To use parentheses, you just overload operator(): template<class T> class Array2D { public: // declarations that will compile T& operator()(int index1, int index2); const T& operator()(int index1, int index2) const; ... }; Clients then use arrays this way: cout << data(3, 6); This is easy to implement and easy to generalize to as many dimensions as you like. The drawback is that your Array2D objects don't look like built-in arrays any more. In fact, the above access to element (3, 6) of data looks, on the face of it, like a function call. If you reject the thought of your arrays looking like FORTRAN refugees, you might turn again to the notion of using brackets as the indexing operator. Although there is no such thing as operator[][], it is nonetheless legal to write code that appears to use it: int data[10][20]; ... cout << data[3][6]; // fine What gives? What gives is that the variable data is not really a two-dimensional array at all, it's a 10-element one-dimensional array. Each of those 10 elements is itself a 20-element array, so the expression data[3][6] really means (data[3])[6], i.e., the seventh element of the array that is the fourth element of data. In short, the value yielded by the first application of the brackets is another array, so the second application of the brackets gets an element from that secondary array. We can play the same game with our Array2D class by overloading operator[] to return an object of a new class, Array1D. We can then overload operator[] again in Array1D to return an element in our original two-dimensional array: template<class T> class Array2D { public: class Array1D { public: T& operator[](int index); const T& operator[](int index) const; ... }; Array1D operator[](int index); const Array1D operator[](int index) const; ... }; The following then becomes legal: Array2D<float> data(10, 20); ... cout << data[3][6]; // fine Here, data[3] yields an Array1D object and the operator[] invocation on that object yields the float in position (3, 6) of the original two-dimensional array. Clients of the Array2D class need not be aware of the presence of the Array1D class. Objects of this latter class stand for one-dimensional array objects that, conceptually, do not exist for clients of Array2D. Such clients program as if they were using real, live, honest-to-Allah two-dimensional arrays. It is of no concern to Array2D clients that those objects must, in order to satisfy the vagaries of C++, be syntactically compatible with one-dimensional arrays of other one-dimensional arrays. ... process them; } The non-const function is a bit more work, because it returns a pointer to a character that may be modified. This is analogous to the behavior of the non-const version of String::operator[] in Item 29, and the implementation is equally analogous: char * String::CharProxy::operator&() { // make sure the character to which this function returns // a pointer isn't shared by any other String objects if (theString.value->isShared()) { theString.value = new StringValue(theString.value->data); } // we don't know how long the pointer this function // returns will be kept by clients, so the StringValue // object can never be shared theString.value->markUnshareable(); return &(theString.value->data[charIndex]); } Much of this code is common to other CharProxy member functions, so I know you'd encapsulate it in a private member function that all would call. A second difference between chars and the CharProxys that stand for them becomes apparent if we have a template for reference-counted arrays that use proxy classes to distinguish lvalue and rvalue invocations of operator[]: template<class T> // reference-counted array class Array { // using proxies public: class Proxy { public: Proxy(Array<T>& array, int index); Proxy& operator=(const T& rhs); operator T() const; ... }; const Proxy operator[](int index) const; Proxy operator[](int index); ... }; Consider how these arrays might be used: Array<int> intArray; ... intArray[5] = 22; // fine intArray[5] += 5; // error! ++intArray[5]; // error! As expected, use of operator[] as the target of a simple assignment succeeds, but use of operator[] on the left-hand side of a call to operator+= or operator++ fails. That's because operator[] returns a proxy object, and there is no operator+= or operator++ for Proxy objects. A similar situation exists for other operators that require lvalues, including operator*=, operator<<=, operator--, etc. If you want these operators to work with operator[] functions that return proxies, you must define each of these functions for the Array<T>::Proxy class. ... HitMap * initializeCollisionMap(); lookup(const string& class1, const string& class2); } // end namespace 246 Item 31 Note the use of the unnamed namespace to contain the functions used to implement processCollision. Everything in such an unnamed namespace is private to the current translation unit (essentially the current file) — it's just like the functions were declared static at file scope. With the advent of namespaces, however, statics at file scope have been deprecated, so you should accustom yourself to using unnamed namespaces as soon as your compilers support them. Conceptually, this implementation is the same as the one that used member functions, but there are some minor differences. First, HitFunctionPtr is now a typedef for a pointer to a non-member function. Second, the exception class CollisionWithUnknownObject has been renamed UnknownCollision and modified to take two objects instead of one. Finally, lookup must now take two type names and perform both parts of the double-dispatch. This means our collision map must now hold three pieces of information: two types names and a HitFunctionPtr. As fate would have it, the standard map class is defined to hold only two pieces of information. We can finesse that problem by using the standard pair template, which lets us bundle the two type names together as a single object. initializeCollisionMap, along with its makeStringPair helper function, then looks like this: // we use this function to create pair<string,string> // objects from two char* literals. It's used in // initializeCollisionMap below. Note how this function // enables the return value optimization (see Item 20). namespace { // unnamed namespace — see below pair<string,string> makeStringPair(const char *s1, const char *s2) { return pair<string,string>(s1, s2); } } // end namespace namespace { // still the unnamed namespace — see below HitMap * initializeCollisionMap() { HitMap *phm = new HitMap; (*phm) [makeStringPair("SpaceShip","Asteroid")] = &shipAsteroid; (*phm)[makeStringPair("SpaceShip", "SpaceStation")] = &shipStation; ... return phm; } } // end namespace Implementing Multiple Dispatch 247 lookup must also be modified to work with the pair objects that now comprise the first component of the collision map: namespace { // I explain this below — trust me HitFunctionPtr lookup(const string& class1, const string& class2) { static auto_ptr collisionMap(initializeCollisionMap()); // see below for a description of make_pair HitMap::iterator mapEntry= collisionMap->find(make_pair(class1, class2)); if (mapEntry == collisionMap->end()) return 0; return (*mapEntry).second; } } // end namespace This is almost exactly what we had before. The only real difference is the use of the make_pair function in this statement: HitMap::iterator mapEntry= collisionMap->find(make_pair(class1, class2)); make_pair is just a convenience function (template) in the standard library (see Item 35) that saves us the trouble of specifying the types when constructing a pair object. We could just as well have written the statement like this: HitMap::iterator mapEntry= collisionMap->find(pair(class1, class2)); This calls for more typing, however, and specifying the types for the pair is redundant (they're the same as the types of class1 and class2), so the make_pair form is more commonly used. Because makeStringPair, initializeCollisionMap, and lookup were declared inside an unnamed namespace, each must be implemented within the same namespace. That's why the implementations of the functions above are in the unnamed namespace (for the same translation unit as their declarations): so the linker will correctly associate their definitions (i.e., their implementations) with their earlier declarations. We have finally achieved our goals. If new subclasses of GameObject are added to our hierarchy, existing classes need not recompile (unless they wish to use the new classes). We have no tangle of RTTI-based switch or if-then-else conditionals to maintain. The addition of new classes to the hierarchy requires only well-defined and localized changes to our system: the addition of one or more map insertions in initializeCollisionMap and the declarations of the new collision-processing functions in the unnamed namespace associated with the implementation of processCollision. It may have been a lot of work to get here, but at least the trip was worthwhile. Yes? Yes? Maybe. Inheritance and Emulated Virtual Function Tables There is one final problem we must confront. (If, at this point, you are wondering if there will always be one final problem to confront, you have truly come to appreciate the difficulty of designing an implementation mechanism for virtual functions.) Everything we've done will work fine as long as we never need to allow inheritance-based type conversions when calling collision-processing functions. But suppose we develop a game in which we must sometimes distinguish between commercial space ships and military space ships. We could modify our hierarchy as follows, where we've heeded the guidance of Item 33 and made the concrete classes CommercialShip and MilitaryShip inherit from the newly abstract class SpaceShip: SpaceShip CommercialShip MilitaryShip GameObject SpaceStation Asteroid Suppose commercial and military ships behave identically when they collide with something. Then we'd expect to be able to use the same collision-processing functions we had before CommercialShip and MilitaryShip were added. In particular, if a MilitaryShip object and an Asteroid collided, we'd expect void shipAsteroid(GameObject& spaceShip, GameObject& asteroid); to be called. It would not be. Instead, an UnknownCollision exception would be thrown. That's because lookup would be asked to find a function corresponding to the type names "MilitaryShip" and "Asteroid," and no such function would be found in collisionMap. Even though a MilitaryShip can be treated like a SpaceShip, lookup has no way of knowing that. Furthermore, there is no easy way of telling it. If you need to implement double-dispatching and you need to support inheritance-based parameter conversions such as these, your only practical recourse is to fall back on the double-virtual-function-call mechanism we examined earlier. That implies you'll also have to put up with everybody recompiling when you add to your inheritance hierarchy, but that's just the way life is sometimes. Initializing Emulated Virtual Function Tables (Reprise) That's really all there is to say about double-dispatching, but it would be unpleasant to end the discussion on such a downbeat note, and unpleasantness is, well, unpleasant. Instead, let's conclude by outlining an alternative approach to initializing collisionMap. As things stand now, our design is entirely static. Once we've registered a function for processing collisions between two types of objects, that's it; we're stuck with that function forever. What if we'd like to add, remove, or change collision-processing functions as the game proceeds? There's no way to do it. But there can be. We can turn the concept of a map for storing collision-processing functions into a class that offers member functions allowing us to modify the contents of the map dynamically. For example: class CollisionMap { public: typedef void (*HitFunctionPtr)(GameObject&, GameObject&); void addEntry(const string& type1, const string& type2, HitFunctionPtr collisionFunction, bool symmetric = true); // see below void removeEntry(const string& type1, const string& type2); HitFunctionPtr lookup(const string& type1, const string& type2); // this function returns a reference to the one and only // map — see Item 26 static CollisionMap& theCollisionMap(); private: // these functions are private to prevent the creation // of multiple maps — see Item 26 CollisionMap(); CollisionMap(const CollisionMap&); }; 250 Item 31 This class lets us add entries to the map, remove them from it, and look up the collision-processing function associated with a particular pair of type names. It also uses the techniques of Item 26 to limit the number of CollisionMap objects to one, because there is only one map in our system. (More complex games with multiple maps are easy to imagine.) Finally, it allows us to simplify the addition of symmetric collisions to the map (i.e., collisions in which the effect of an object of type T1 hitting an object of type T2 is the same as that of an object of type T2 hitting an object of type T1) by automatically adding the implied map entry when addEntry is called with the optional parameter symmetric set to true. With the CollisionMap class, each client wishing to add an entry to the map does so directly: void shipAsteroid(GameObject& spaceShip, GameObject& asteroid); CollisionMap::theCollisionMap().addEntry("SpaceShip", "Asteroid", &shipAsteroid); void shipStation(GameObject& spaceShip, GameObject& spaceStation); CollisionMap::theCollisionMap().addEntry("SpaceShip", "SpaceStation", &shipStation); void asteroidStation(GameObject& asteroid, GameObject& spaceStation); CollisionMap::theCollisionMap().addEntry("Asteroid", "SpaceStation", &asteroidStation); ... Care must be taken to ensure that these map entries are added to the map before any collisions occur that would call the associated functions. One way to do this would be to have constructors in GameObject subclasses check to make sure the appropriate mappings had been added each time an object was created. Such an approach would exact a small performance penalty at runtime. An alternative would be to create a RegisterCollisionFunction class: class RegisterCollisionFunction { public: RegisterCollisionFunction(const string& type1, const string& type2, CollisionMap::HitFunctionPtr collisionFunction, bool symmetric = true) { CollisionMap::theCollisionMap().addEntry(type1, type2, collisionFunction, symmetric); } }; Implementing Multiple Dispatch 251 Clients could then use global objects of this type to automatically register the functions they need: RegisterCollisionFunction cf1("SpaceShip", "Asteroid", &shipAsteroid); RegisterCollisionFunction cf2("SpaceShip", "SpaceStation", &shipStation); RegisterCollisionFunction cf3("Asteroid", "SpaceStation", &asteroidStation); ... int main(int argc, char * argv[]) { ... } Because these objects are created before main is invoked, the functions their constructors register are also added to the map before main is called. If, later, a new derived class is added class Satellite: public GameObject { ... }; and one or more new collision-processing functions are written, void satelliteShip(GameObject& satellite, GameObject& spaceShip); void satelliteAsteroid(GameObject& satellite, GameObject& asteroid); these new functions can be similarly added to the map without disturbing existing code: RegisterCollisionFunction cf4("Satellite", "SpaceShip", &satelliteShip); RegisterCollisionFunction cf5("Satellite", "Asteroid", &satelliteAsteroid); This doesn't change the fact that there's no perfect way to implement multiple dispatch, but it does make it easy to provide data for a map-based implementation if we decide such an approach is the best match for our needs. Miscellany We thus arrive at the organizational back of the bus, the chapter containing the guidelines no one else would have. We begin with two Items on C++ software development that describe how to design systems that accommodate change. One of the strengths of the object-oriented approach to systems building is its support for change, and these Items describe specific steps you can take to fortify your software against the slings and arrows of a world that refuses to stand still. Miscellany We then examine how to combine C and C++ in the same program. This necessarily leads to consideration of extralinguistic issues, but C++ exists in the real world, so sometimes we must confront such things. Finally, I summarize changes to the C++ language standard since publication of the de facto reference. I especially cover the sweeping changes that have been made in the standard library. If you have not been following the standardization process closely, you are probably in for some surprises — many of them quite pleasant. Item 32: Program in the future tense. Programming in the Future Tense As software developers, we may not know much, but we do know that things will change. We don't necessarily know what will change, how the changes will be brought about, when the changes will occur, or why they will take place, but we do know this: things will change. Good software adapts well to change. It accommodates new features, it ports to new platforms, it adjusts to new demands, it handles new inputs. Software this flexible, this robust, and this reliable does not come about by accident. It is designed and implemented by programmers who conform to the constraints of today while keeping in mind the probable needs of tomorrow. This kind of software — software that Programming in the Future Tense 253 accepts change gracefully — is written by people who program in the future tense. To program in the future tense is to accept that things will change and to be prepared for it. It is to recognize that new functions will be added to libraries, that new overloadings will occur, and to watch for the potentially ambiguous function calls that might result. It is to acknowledge that new classes will be added to hierarchies, that present-day derived classes may be tomorrow's base classes, and to prepare for that possibility. It is to accept that new applications will be written, that functions will be called in new contexts, and to write those functions so they continue to perform correctly. It is to remember that the programmers charged with software maintenance are typically not the code's original developers, hence to design and implement in a fashion that facilitates comprehension, modification, and enhancement by others. One way to do this is to express design constraints in C++ instead of (or in addition to) comments or other documentation. For example, if a class is designed to never have derived classes, don't just put a comment in the header file above the class, use C++ to prevent derivation; Item 26 shows you how. If a class requires that all instances be on the heap, don't just tell clients that, enforce the restriction by applying the approach of Item 27. If copying and assignment make no sense for a class, prevent those operations by declaring the copy constructor and the assignment operator private. C++ offers great power, flexibility, and expressiveness. Use these characteristics of the language to enforce the design decisions in your programs. Given that things will change, write classes that can withstand the rough-and-tumble world of software evolution. Avoid "demand-paged" virtual functions, whereby you make no functions virtual unless somebody comes along and demands that you do it. Instead, determine the meaning of a function and whether it makes sense to let it be redefined in derived classes. If it does, declare it virtual, even if nobody redefines it right away. If it doesn't, declare it nonvirtual, and don't change it later just because it would be convenient for someone; make sure the change makes sense in the context of the entire class and the abstraction it represents. Handle assignment and copy construction in every class, even if "nobody ever does those things." Just because they don't do them now doesn't mean they won't do them in the future. If these functions are difficult to implement, declare them private. That way no one will inadvertently call compiler-generated functions that do the wrong thing (as often happens with default assignment operators and copy constructors). 254 Item 32 Adhere to the principle of least astonishment: strive to provide classes whose operators and functions have a natural syntax and an intuitive semantics. Preserve consistency with the behavior of the built-in types: when in doubt, do as the ints do. Recognize that anything somebody can do, they will do. They'll throw exceptions, they'll assign objects to themselves, they'll use objects before giving them values, they'll give objects values and never use them, they'll give them huge values, they'll give them tiny values, they'll give them null values. In general, if it will compile, somebody will do it. As a result, make your classes easy to use correctly and hard to use incorrectly. Accept that clients will make mistakes, and design your classes so you can prevent, detect, or correct such errors (see, for example, Item 33). Strive for portable code. It's not much harder to write portable programs than to write unportable ones, and only rarely will the difference in performance be significant enough to justify unportable constructs (see Item 16). Even programs designed for custom hardware often end up being ported, because stock hardware generally achieves an equivalent level of performance within a few years. Writing portable code allows you to switch platforms easily, to enlarge your client base, and to brag about supporting open systems. It also makes it easier to recover if you bet wrong in the operating system sweepstakes. Design your code so that when changes are necessary, the impact is localized. Encapsulate as much as you can; make implementation details private. Where applicable, use unnamed namespaces or file-static objects and functions (see Item 31). Try to avoid designs that lead to virtual base classes, because such classes must be initialized by every class derived from them — even those derived indirectly (see Item 4). Avoid RTTI-based designs that make use of cascading if-then-else statements (see Item 31 again). Every time the class hierarchy changes, each set of statements must be updated, and if you forget one, you'll receive no warning from your compilers. These are well known and oft-repeated exhortations, but most programmers are still stuck in the present tense. As are many authors, unfortunately. Consider this advice by a well-regarded C++ expert: You need a virtual destructor whenever someone deletes a B* that actually points to a D. Here B is a base class and D is a derived class. In other words, this author suggests that if your program looks like this, you don't need a virtual destructor in B: Programming in the Future Tense 255 class B { ... }; // no virtual dtor class D: public B { ... }; B *pb = new D; However, the situation changes if you add this statement: delete pb; // NOW you need the virtual // destructor in B The implication is that a minor change to client code — the addition of a delete statement — can result in the need to change the class definition for B. When that happens, all B's clients must recompile. Following this author's advice, then, the addition of a single statement in one function can lead to extensive code recompilation and relinking for all clients of a library. This is anything but effective software design. On the same topic, a different author writes: Consider this code: class string { // from the standard C++ library public: ~string(); }; class B { ... }; // no data members with dtors, // no virtual dtor needed But if a new class is derived from B, things change: class D: public B { string name; // NOW ~B needs to be virtual }; Again, a small change to the way B is used (here, the addition of a derived class that contains a member with a destructor) may necessitate extensive recompilation and relinking by clients. But small changes in software should have small impacts on systems. This design fails that test. The same author writes: If a multiple inheritance hierarchy has any destructors, every base class should have a virtual destructor. In all these quotations, note the present-tense thinking. How do clients manipulate pointers now? What class members have destructors now? What classes in the hierarchy have destructors now? 256 Item 32 Future-tense thinking is quite different. Instead of asking how a class is used now, it asks how the class is designed to be used. Future-tense thinking says, if a class is designed to be used as a base class (even if it's not used as one now), it should have a virtual destructor. Such classes behave correctly both now and in the future, and they don't affect other library clients when new classes derive from them. (At least, they have no effect as far as their destructor is concerned. If additional changes to the class are required, other clients may be affected.) A commercial class library (one that predates the string specification in the C++ library standard) contains a string class with no virtual destructor. The vendor's explanation? We didn't make the destructor virtual, because we didn't want String to have a vtbl. We have no intention of ever having a String*, so this is not a problem. We are well aware of the difficulties this could cause. Is this present-tense or future-tense thinking? Certainly the vtbl issue is a legitimate technical concern (see Item 24). The implementation of most String classes contains only a single char* pointer inside each String object, so adding a vptr to each String would double the size of those objects. It is easy to understand why a vendor would be unwilling to do that, especially for a highly visible, heavily used class like String. The performance of such a class might easily fall within the 20% of a program that makes a difference (see Item 16). Still, the total memory devoted to a string object — the memory for the object itself plus the heap memory needed to hold the string's value — is typically much greater than just the space needed to hold a char* pointer. From this perspective, the overhead imposed by a vptr is less significant. Nevertheless, it is a legitimate technical consideration. (Certainly the ISO/ANSI standardization committee seems to think so: the standard string type has a nonvirtual destructor.) Somewhat more troubling is the vendor's remark, "We have no intention of ever having a String*, so this is not a problem." That may be true, but their String class is part of a library they make available to thousands of developers. That's a lot of developers, each with a different level of experience with C++, each doing something unique. Do those developers understand the consequences of there being no virtual destructor in String? Are they likely to know that because String has no virtual destructor, deriving new classes from String is a high-risk venture? Is this vendor confident their clients will understand that in the absence of a virtual destructor, deleting objects through String* pointers will not work properly and RTTI operations on Programming in the Future Tense 257 pointers and references to Strings may return incorrect information? Is this class easy to use correctly and hard to use incorrectly? This vendor should provide documentation for its String class that makes clear the class is not designed for derivation, but what if programmers overlook the caveat or flat-out fail to read the documentation? An alternative would be to prohibit derivation. Item 26 describes how to do this by limiting object creation to the heap and then using auto_ptr objects to manipulate the heap objects. The interface for String creation would then be both unconventional and inconvenient, requiring this, auto_ptr ps(String::makeString("Future tense C++")); ... // treat ps as a pointer to // a String object, but don't // worry about deleting it instead of this, String s("Future tense C++"); but perhaps the reduction in the risk of improperly behaving derived classes would be worth the syntactic inconvenience. (For String, this is unlikely to be the case, but for other classes, the trade-off might well be worth it.) There is a need, of course, for present-tense thinking. The software you're developing has to work with current compilers; you can't afford to wait until the latest language features are implemented. It has to run on the hardware you currently support and it must do so under configurations your clients have available; you can't force your customers to upgrade their systems or modify their operating environment. It has to offer acceptable performance now; promises of smaller, faster programs some years down the line don't generally warm the cockles of potential customers' hearts. And the software you're working on must be available "soon," which often means some time in the recent past. These are important constraints. You cannot ignore them. Future-tense thinking simply adds a few additional considerations: ■ Provide complete classes, even if some parts aren't currently used. When new demands are made on your classes, you're less likely to have to go back and modify them. 258 Item 33 ■ Design your interfaces to facilitate common operations and prevent common errors. Make the classes easy to use correctly, hard to use incorrectly. For example, prohibit copying and assignment for classes where those operations make no sense. Prevent partial assignments (see Item 33). ■ If there is no great penalty for generalizing your code, generalize it. For example, if you are writing an algorithm for tree traversal, consider generalizing it to handle any kind of directed acyclic graph. Future tense thinking increases the reusability of the code you write, enhances its maintainability, makes it more robust, and facilitates graceful change in an environment where change is a certainty. It must be balanced against present-tense constraints. Too many programmers focus exclusively on current needs, however, and in so doing they sacrifice the long-term viability of the software they design and implement. Be different. Be a renegade. Program in the future tense. Item 33: Make non-leaf classes abstract. Suppose you're working on a project whose software deals with animals. Within this software, most animals can be treated pretty much the same, but two kinds of animals — lizards and chickens — require special handling. That being the case, the obvious way to relate the classes for animals, lizards, and chickens is like this: Making Non-Leaf Classes Abstract Chicken The Animal class embodies the features shared by all the creatures you deal with, and the Lizard and Chicken classes specialize Animal in ways appropriate for lizards and chickens, respectively. Here's a sketch of the definitions for these classes: class Animal { public: Animal& operator=(const Animal& rhs); ... }; Making Non-Leaf Classes Abstract 259 class Lizard: public Animal { public: Lizard& operator=(const Lizard& rhs); ... }; class Chicken: public Animal { public: Chicken& operator=(const Chicken& rhs); ... }; Only the assignment operators are shown here, but that's more than enough to keep us busy for a while. Consider this code: Lizard liz1; Lizard liz2; Animal *pAnimal1 = &liz1; Animal *pAnimal2 = &liz2; ... *pAnimal1 = *pAnimal2; There are two problems here. First, the assignment operator invoked on the last line is that of the Animal class, even though the objects involved are of type Lizard. As a result, only the Animal part of liz1 will be modified. This is a partial assignment. After the assignment, liz1's Animal members have the values they got from liz2, but liz1's Lizard members remain unchanged. The second problem is that real programmers write code like this. It's not uncommon to make assignments to objects through pointers, especially for experienced C programmers who have moved to C++. That being the case, we'd like to make the assignment behave in a more reasonable fashion. As Item 32 points out, our classes should be easy to use correctly and difficult to use incorrectly, and the classes in the hierarchy above are easy to use incorrectly. One approach to the problem is to make the assignment operators virtual. If Animal::operator= were virtual, the assignment would invoke the Lizard assignment operator, which is certainly the correct one to call. However, look what happens if we declare the assignment operators virtual: class Animal { public: virtual Animal& operator=(const Animal& rhs); ... }; 260 Item 33 class Lizard: public Animal { public: virtual Lizard& operator=(const Animal& rhs); ... }; class Chicken: public Animal { public: virtual Chicken& operator=(const Animal& rhs); ... }; Due to relatively recent changes to the language, we can customize the return value of the assignment operators so that each returns a reference to the correct class, but the rules of C++ force us to declare identical parameter types for a virtual function in every class in which it is declared. That means the assignment operator for the Lizard and Chicken classes must be prepared to accept any kind of Animal object on the right-hand side of an assignment. That, in turn, means we have to confront the fact that code like the following is legal: Lizard liz; Chicken chick; Animal *pAnimal1 = &liz; Animal *pAnimal2 = &chick; ... *pAnimal1 = *pAnimal2; // assign a chicken to // a lizard! This is a mixed-type assignment: a Lizard is on the left and a Chicken is on the right. Mixed-type assignments aren't usually a problem in C++, because the language's strong typing generally renders them illegal. By making Animal's assignment operator virtual, however, we opened the door to such mixed-type operations. This puts us in a difficult position. We'd like to allow same-type assignments through pointers, but we'd like to forbid mixed-type assignments through those same pointers. In other words, we want to allow this, Lizard liz1, liz2; Animal *pAnimal1 = &liz1; Animal *pAnimal2 = &liz2; ... *pAnimal1 = *pAnimal2; // assign a lizard to a lizard Making Non-Leaf Classes Abstract 261 but we want to prohibit this: Animal *pAnimal1 = &liz; Animal *pAnimal2 = &chick; ... *pAnimal1 = *pAnimal2; // assign a chicken to a lizard Distinctions such as these can be made only at runtime, because sometimes assigning *pAnimal2 to *pAnimal1 is valid, sometimes it's not. We thus enter the murky world of type-based runtime errors. In particular, we need to signal an error inside operator= if we're faced with a mixed-type assignment, but if the types are the same, we want to perform the assignment in the usual fashion. We can use a dynamic_cast (see Item 2) to implement this behavior. Here's how to do it for Lizard's assignment operator: Lizard& Lizard::operator=(const Animal& rhs) { // make sure rhs is really a lizard const Lizard& rhs_liz = dynamic_cast(rhs); proceed with a normal assignment of rhs_liz to *this; } This function assigns rhs to *this only if rhs is really a Lizard. If it's not, the dynamic_cast will throw a bad_cast exception. (Actually, the type of the exception is std::bad_cast, because the components of the standard library, including the exceptions thrown by the standard components, are in the namespace std. For an overview of the standard library, see Item 35.) Even without worrying about exceptions, this function seems needlessly complicated and expensive — the dynamic_cast must consult a type_info structure; see Item 24 — in the common case where one Lizard object is assigned to another: Lizard liz1, liz2; ... liz1 = liz2; // no need to perform a // dynamic_cast: this // assignment must be valid We can handle this case without paying for the complexity or cost of a dynamic_cast by adding to Lizard the conventional assignment operator: 262 Item 33 class Lizard: public Animal { public: virtual Lizard& operator=(const Animal& rhs); Lizard& operator=(const Lizard& rhs); // add this ... }; Lizard liz1, liz2; ... liz1 = liz2; // calls operator= taking // a const Lizard& Animal *pAnimal1 = &liz1; Animal *pAnimal2 = &liz2; ... *pAnimal1 = *pAnimal2; // calls operator= taking // a const Animal& In fact, given this latter operator=, it's simplicity itself to implement the former one in terms of it: Lizard& Lizard::operator=(const Animal& rhs) { return operator=(dynamic_cast(rhs)); } This function attempts to cast rhs to be a Lizard. If the cast succeeds, the normal class assignment operator is called. Otherwise, a bad_cast exception is thrown. Frankly, all this business of checking types at runtime and using dynamic_casts makes me nervous. For one thing, some compilers still lack support for dynamic_cast, so code that uses it, though theoretically portable, is not necessarily portable in practice. More importantly, it requires that clients of Lizard and Chicken be prepared to catch bad_cast exceptions and do something sensible with them each time they perform an assignment. In my experience, there just aren't that many programmers who are willing to program that way. If they don't, it's not clear we've gained a whole lot over our original situation where we were trying to guard against partial assignments. Given this rather unsatisfactory state of affairs regarding virtual assignment operators, it makes sense to regroup and try to find a way to prevent clients from making problematic assignments in the first place. If such assignments are rejected during compilation, we don't have to worry about them doing the wrong thing. Making Non-Leaf Classes Abstract 263 The easiest way to prevent such assignments is to make operator= private in Animal. That way, lizards can be assigned to lizards and chickens can be assigned to chickens, but partial and mixed-type assignments are forbidden: class Animal { private: Animal& operator=(const Animal& rhs); // this is now ... // private }; class Lizard: public Animal { public: Lizard& operator=(const Lizard& rhs); ... }; class Chicken: public Animal { public: Chicken& operator=(const Chicken& rhs); ... }; Lizard liz1, liz2; ... liz1 = liz2; // fine Chicken chick1, chick2; ... chick1 = chick2; // also fine Animal *pAnimal1 = &liz1; Animal *pAnimal2 = &chick1; ... *pAnimal1 = *pAnimal2; // error! attempt to call // private Animal::operator= Unfortunately, Animal is a concrete class, and this approach also makes assignments between Animal objects illegal: Animal animal1, animal2; ... animal1 = animal2; // error! attempt to call // private Animal::operator= Moreover, it makes it impossible to implement the Lizard and Chicken assignment operators correctly, because assignment operators in derived classes are responsible for calling assignment operators in their base classes: 264 Item 33 Lizard& Lizard::operator=(const Lizard& rhs) { if (this == &rhs) return *this; Animal::operator=(rhs); // error! attempt to call // private function. But // Lizard::operator= must // call this function to ... // assign the Animal parts } // of *this! We can solve this latter problem by declaring Animal::operator= protected, but the conundrum of allowing assignments between Animal objects while preventing partial assignments of Lizard and Chicken objects through Animal pointers remains. What's a poor programmer to do? The easiest thing is to eliminate the need to allow assignments between Animal objects, and the easiest way to do that is to make Animal an abstract class. As an abstract class, Animal can't be instantiated, so there will be no need to allow assignments between Animals. Of course, this leads to a new problem, because our original design for this system presupposed that Animal objects were necessary. There is an easy way around this difficulty. Instead of making Animal itself abstract, we create a new class — AbstractAnimal, say — consisting of the common features of Animal, Lizard, and Chicken objects, and we make that class abstract. Then we have each of our concrete classes inherit from AbstractAnimal. The revised hierarchy looks like this, AbstractAnimal Animal Lizard Chicken and the class definitions are as follows: class AbstractAnimal { protected: AbstractAnimal& operator=(const AbstractAnimal& rhs); public: virtual ~AbstractAnimal() = 0; // see below ... }; Making Non-Leaf Classes Abstract 265 class Animal: public AbstractAnimal { public: Animal& operator=(const Animal& rhs); ... }; class Lizard: public AbstractAnimal { public: Lizard& operator=(const Lizard& rhs); ... }; class Chicken: public AbstractAnimal { public: Chicken& operator=(const Chicken& rhs); ... }; This design gives you everything you need. Homogeneous assignments are allowed for lizards, chickens, and animals; partial assignments and heterogeneous assignments are prohibited; and derived class assignment operators may call the assignment operator in the base class. Furthermore, none of the code written in terms of the Animal, Lizard, or Chicken classes requires modification, because these classes continue to exist and to behave as they did before AbstractAnimal was introduced. Sure, such code has to be recompiled, but that's a small price to pay for the security of knowing that assignments that compile will behave intuitively and assignments that would behave unintuitively won't compile. For all this to work, AbstractAnimal must be abstract — it must contain at least one pure virtual function. In most cases, coming up with a suitable function is not a problem, but on rare occasions you may find yourself facing the need to create a class like AbstractAnimal in which none of the member functions would naturally be declared pure virtual. In such cases, the conventional technique is to make the destructor a pure virtual function; that's what's shown above. In order to support polymorphism through pointers correctly, base classes need virtual destructors anyway, so the only cost associated with making such destructors pure virtual is the inconvenience of having to implement them outside their class definitions. (For an example, see page 195.) (If the notion of implementing a pure virtual function strikes you as odd, you just haven't been getting out enough. Declaring a function pure virtual doesn't mean it has no implementation, it means ■ the current class is abstract, and ■ any concrete class inheriting from the current class must declare the function as a "normal" virtual function (i.e., without the "=0"). 266 Item 33 True, most pure virtual functions are never implemented, but pure virtual destructors are a special case. They must be implemented, because they are called whenever a derived class destructor is invoked. Furthermore, they often perform useful tasks, such as releasing resources (see Item 9) or logging messages. Implementing pure virtual functions may be uncommon in general, but for pure virtual destructors, it's not just common, it's mandatory.) You may have noticed that this discussion of assignment through base class pointers is based on the assumption that concrete derived classes like Lizard contain data members. If there are no data members in a derived class, you might point out, there is no problem, and it would be safe to have a dataless concrete class inherit from another concrete class. However, just because a class has no data now is no reason to conclude that it will have no data in the future. If it might have data members in the future, all you're doing is postponing the problem until the data members are added, in which case you're merely trading short-term convenience for long-term grief (see also Item 32). Replacement of a concrete base class like Animal with an abstract base class like AbstractAnimal yields benefits far beyond simply making the behavior of operator= easier to understand. It also reduces the chances that you'll try to treat arrays polymorphically, the unpleasant consequences of which are examined in Item 3. The most significant benefit of the technique, however, occurs at the design level, because replacing concrete base classes with abstract base classes forces you to explicitly recognize the existence of useful abstractions. That is, it makes you create new abstract classes for useful concepts, even if you aren't aware of the fact that the useful concepts exist. If you have two concrete classes C1 and C2 and you'd like C2 to publicly inherit from C1, you should transform that two-class hierarchy into a three-class hierarchy by creating a new abstract class A and having both C1 and C2 publicly inherit from it: C1 C2 Your initial idea A C1 C2 Your transformed hierarchy Making Non-Leaf Classes Abstract 267 The primary value of this transformation is that it forces you to identify the abstract class A. Clearly, C1 and C2 have something in common; that's why they're related by public inheritance. With this transformation, you must identify what that something is. Furthermore, you must formalize the something as a class in C++, at which point it becomes more than just a vague something, it achieves the status of a formal abstraction, one with well-defined member functions and well-defined semantics. All of which leads to some worrisome thinking. After all, every class represents some kind of abstraction, so shouldn't we create two classes for every concept in our hierarchy, one being abstract (to embody the abstract part of the abstraction) and one being concrete (to embody the object-generation part of the abstraction)? No. If you do, you'll end up with a hierarchy with too many classes. Such a hierarchy is difficult to understand, hard to maintain, and expensive to compile. That is not the goal of object-oriented design. The goal is to identify useful abstractions and to force them — and only them — into existence as abstract classes. But how do you identify useful abstractions? Who knows what abstractions might prove useful in the future? Who can predict who's going to want to inherit from what? Well, I don't know how to predict the future uses of an inheritance hierarchy, but I do know one thing: the need for an abstraction in one context may be coincidental, but the need for an abstraction in more than one context is usually meaningful. Useful abstractions, then, are those that are needed in more than one context. That is, they correspond to classes that are useful in their own right (i.e., it is useful to have objects of that type) and that are also useful for purposes of one or more derived classes. This is precisely why the transformation from concrete base class to abstract base class is useful: it forces the introduction of a new abstract class only when an existing concrete class is about to be used as a base class, i.e., when the class is about to be (re)used in a new context. Such abstractions are useful, because they have, through demonstrated need, shown themselves to be so. The first time a concept is needed, we can't justify the creation of both an abstract class (for the concept) and a concrete class (for the objects corresponding to that concept), but the second time that concept is needed, we can justify the creation of both the abstract and the concrete classes. The transformation I've described simply mechanizes this process, and in so doing it forces designers and programmers to represent explicitly those abstractions that are useful, even if the de- 268 Item 33 signers and programmers are not consciously aware of the useful concepts. It also happens to make it a lot easier to bring sanity to the behavior of assignment operators. Let's consider a brief example. Suppose you're working on an application that deals with moving information between computers on a network by breaking it into packets and transmitting them according to some protocol. All we'll consider here is the class or classes for representing packets. We'll assume such classes make sense for this application. Suppose you deal with only a single kind of transfer protocol and only a single kind of packet. Perhaps you've heard that other protocols and packet types exist, but you've never supported them, nor do you have any plans to support them in the future. Should you make an abstract class for packets (for the concept that a packet represents) as well as a concrete class for the packets you'll actually be using? If you do, you could hope to add new packet types later without changing the base class for packets. That would save you from having to recompile packet-using applications if you add new packet types. But that design requires two classes, and right now you need only one (for the particular type of packets you use). Is it worth complicating your design now to allow for future extension that may never take place? There is no unequivocally correct choice to be made here, but experience has shown it is nearly impossible to design good classes for concepts we do not understand well. If you create an abstract class for packets, how likely are you to get it right, especially since your experience is limited to only a single packet type? Remember that you gain the benefit of an abstract class for packets only if you can design that class so that future classes can inherit from it without its being changed in any way. (If it needs to be changed, you have to recompile all packet clients, and you've gained nothing.) It is unlikely you could design a satisfactory abstract packet class unless you were well versed in many different kinds of packets and in the varied contexts in which they are used. Given your limited experience in this case, my advice would be not to define an abstract class for packets, adding one later only if you find a need to inherit from the concrete packet class. The transformation I've described here is a way to identify the need for abstract classes, not the way. There are many other ways to identify good candidates for abstract classes; books on object-oriented analysis are filled with them. It's not the case that the only time you should introduce abstract classes is when you find yourself wanting to have a concrete class inherit from another concrete class. However, the desire to relate two concrete classes by public inheritance is usually indicative of a need for a new abstract class. As is often the case in such matters, brash reality sometimes intrudes on the peaceful ruminations of theory. Third-party C++ class libraries are proliferating with gusto, and what are you to do if you find yourself wanting to create a concrete class that inherits from a concrete class in a library to which you have only read access? You can't modify the library to insert a new abstract class, so your choices are both limited and unappealing: Making Non-Leaf Classes Abstract 269 ■ Derive your concrete class from the existing concrete class, and put up with the assignment-related problems we examined at the beginning of this Item. You'll also have to watch out for the array-related pitfalls described in Item 3. ■ Try to find an abstract class higher in the library hierarchy that does most of what you need, then inherit from that class. Of course, there may not be a suitable class, and even if there is, you may have to duplicate a lot of effort that has already been put into the implementation of the concrete class whose functionality you'd like to extend. ■ Implement your new class in terms of the library class you'd like to inherit from. For example, you could have an object of the library class as a data member, then reimplement the library class's interface in your new class: class Window { // this is the library class public: virtual void resize(int newWidth, int newHeight); virtual void repaint() const; int width() const; int height() const; }; class SpecialWindow { // this is the class you public: // wanted to have inherit ... // from Window // pass-through implementations of nonvirtual functions int width() const { return w.width(); } int height() const { return w.height(); } // new implementations of "inherited" virtual functions virtual void resize(int newWidth, int newHeight); virtual void repaint() const; private: Window w; }; 270 Item 34 This strategy requires that you be prepared to update your class each time the library vendor updates the class on which you're dependent. It also requires that you be willing to forgo the ability to redefine virtual functions declared in the library class, because you can't redefine virtual functions unless you inherit them. ■ Make do with what you've got. Use the concrete class that's in the library and modify your software so that the class suffices. Write non-member functions to provide the functionality you'd like to add to the class, but can't. The resulting software may not be as clear, as efficient, as maintainable, or as extensible as you'd like, but at least it will get the job done. None of these choices is particularly attractive, so you have to apply some engineering judgment and choose the poison you find least unappealing. It's not much fun, but life's like that sometimes. To make things easier for yourself (and the rest of us) in the future, complain to the vendors of libraries whose designs you find wanting. With luck (and a lot of comments from clients), those designs will improve as time goes on. Still, the general rule remains: non-leaf classes should be abstract. You may need to bend the rule when working with outside libraries, but in code over which you have control, adherence to it will yield dividends in the form of increased reliability, robustness, comprehensibility, and extensibility throughout your software. Item 34: Understand how to combine C++ and C in the same program. In many ways, the things you have to worry about when making a program out of some components in C++ and some in C are the same as those you have to worry about when cobbling together a C program out of object files produced by more than one C compiler. There is no way to combine such files unless the different compilers agree on implementation-dependent features like the size of ints and doubles, the mechanism by which parameters are passed from caller to callee, and whether the caller or the callee orchestrates the passing. These pragmatic aspects of mixed-compiler software development are quite properly ignored by language standardization efforts, so the only reliable way to know that object files from compilers A and B can be safely combined in a program is to obtain assurances from the vendors of A and B that their products produce compatible output. This is as true for programs made up of C++ and C as it is for all-C++ or all-C programs, so before you try to mix C++ and C in the same program, make sure your C++ and C compilers generate compatible object files. Combining C++ and C in the Same Program 271 Having done that, there are four other things you need to consider: name mangling, initialization of statics, dynamic memory allocation, and data structure compatibility. Name Mangling Name mangling, as you may know, is the process through which your C++ compilers give each function in your program a unique name. In C, this process is unnecessary, because you can't overload function names, but nearly all C++ programs have at least a few functions with the same name. (Consider, for example, the iostream library, which declares several versions of operator<< and operator>>.) Overloading is incompatible with most linkers, because linkers generally take a dim view of multiple functions with the same name. Name mangling is a concession to the realities of linkers; in particular, to the fact that linkers usually insist on all function names being unique. As long as you stay within the confines of C++, name mangling is not likely to concern you. If you have a function name drawLine that a compiler mangles into xyzzy, you'll always use the name drawLine, and you'll have little reason to care that the underlying object files happen to refer to xyzzy. It's a different story if drawLine is in a C library. In that case, your C++ source file probably includes a header file that contains a declaration like this, void drawLine(int x1, int y1, int x2, int y2); and your code contains calls to drawLine in the usual fashion. Each such call is translated by your compilers into a call to the mangled name of that function, so when you write this, drawLine(a, b, c, d); // call to unmangled function name your object files contain a function call that corresponds to this: xyzzy(a, b, c, d); // call to mangled function name But if drawLine is a C function, the object file (or archive or dynamically linked library, etc.) that contains the compiled version of drawLine contains a function called drawLine; no name mangling has taken place. When you try to link the object files comprising your program together, you'll get an error, because the linker is looking for a function called xyzzy, and there is no such function. To solve this problem, you need a way to tell your C++ compilers not to mangle certain function names. You never want to mangle the names of functions written in other languages, whether they be in C, assembler, FORTRAN, Lisp, Forth, or what-have-you. (Yes, what-have-you 272 Item 34 would include COBOL, but then what would you have?) After all, if you call a C function named drawLine, it's really called drawLine, and your object code should contain a reference to that name, not to some mangled version of that name. To suppress name mangling, use C++'s extern "C" directive: // declare a function called drawLine; don't mangle // its name extern "C" void drawLine(int x1, int y1, int x2, int y2); Don't be drawn into the trap of assuming that where there's an extern "C", there must be an extern "Pascal" and an extern "FORTRAN" as well. There's not, at least not in the standard. The best way to view extern "C" is not as an assertion that the associated function is written in C, but as a statement that the function should be called as if it were written in C. (Technically, extern "C" means the function has C linkage, but what that means is far from clear. One thing it always means, however, is that name mangling is suppressed.) For example, if you were so unfortunate as to have to write a function in assembler, you could declare it extern "C", too: // this function is in assembler — don't mangle its name extern "C" void twiddleBits(unsigned char bits); You can even declare C++ functions extern "C". This can be useful if you're writing a library in C++ that you'd like to provide to clients using other programming languages. By suppressing the name mangling of your C++ function names, your clients can use the natural and intuitive names you choose instead of the mangled names your compilers would otherwise generate: // the following C++ function is designed for use outside // C++ and should not have its name mangled extern "C" void simulate(int iterations); Often you'll have a slew of functions whose names you don't want mangled, and it would be a pain to precede each with extern "C". Fortunately, you don't have to. extern "C" can also be made to apply to a whole set of functions. Just enclose them all in curly braces: extern "C" { // disable name mangling for // all the following functions void drawLine(int x1, int y1, int x2, int y2); void twiddleBits(unsigned char bits); void simulate(int iterations); ... } Combining C++ and C in the Same Program 273 This use of extern "C" simplifies the maintenance of header files that must be used with both C++ and C. When compiling for C++, you'll want to include extern "C", but when compiling for C, you won't. By taking advantage of the fact that the preprocessor symbol __cplusplus is defined only for C++ compilations, you can structure your polyglot header files as follows: #ifdef __cplusplus extern "C" { #endif void drawLine(int x1, int y1, int x2, int y2); void twiddleBits(unsigned char bits); void simulate(int iterations); ... #ifdef __cplusplus } #endif There's no such thing as a "standard" name mangling algorithm, by the way. Different compilers are free to mangle names in different ways, and different compilers do. This is a good thing. If all compilers mangled names the same way, you might be lulled into thinking they all generated compatible code. The way things are now, if you try to mix object code from incompatible C++ compilers, there's a good chance you'll get an error during linking, because the mangled names won't match up. This implies you'll probably have other compatibility problems, too, and it's better to find out about such incompatibilities sooner than later. Initialization of Statics Once you've mastered name mangling, you need to deal with the fact that in C++, lots of code can get executed before and after main. In particular, the constructors of static class objects and objects at global, namespace, and file scope are usually called before the body of main is executed. This process is known as static initialization. This is in direct opposition to the way we normally think about C++ and C programs, in which we view main as the entry point to execution of the program. Similarly, objects that are created through static initialization must have their destructors called during static destruction; that process typically takes place after main has finished executing. To resolve the dilemma that main is supposed to be invoked first, yet objects need to be constructed before main is executed, many compilers insert a call to a special compiler-written function at the beginning of 274 Item 34 main, and it is this special function that takes care of static initialization. Similarly, compilers often insert a call to another special function at the end of main to take care of the destruction of static objects. Code generated for main often looks as if main had been written like this: int main(int argc, char *argv[]) { performStaticInitialization(); // generated by the // implementation the statements you put in main go here; performStaticDestruction(); // generated by the // implementation } Now don't take this too literally. The functions performStaticInitialization and performStaticDestruction usually have much more cryptic names, and they may even be generated inline, in which case you won't see any functions for them in your object files. The important point is this: if a C++ compiler adopts this approach to the initialization and destruction of static objects, such objects will be neither initialized nor destroyed unless main is written in C++. Because this approach to static initialization and destruction is common, you should try to write main in C++ if you write any part of a software system in C++. Sometimes it would seem to make more sense to write main in C — say if most of a program is in C and C++ is just a support library. Nevertheless, there's a good chance the C++ library contains static objects (if it doesn't now, it probably will in the future — see Item 32), so it's still a good idea to write main in C++ if you possibly can. That doesn't mean you need to rewrite your C code, however. Just rename the main you wrote in C to be realMain, then have the C++ version of main call realMain: extern "C" // implement this int realMain(int argc, char *argv[]); // function in C int main(int argc, char *argv[]) // write this in C++ { return realMain(argc, argv); } If you do this, it's a good idea to put a comment above main explaining what is going on. If you cannot write main in C++, you've got a problem, because there is no other portable way to ensure that constructors and destructors for static objects are called. This doesn't mean all is lost, it just means you'll have to work a little harder. Compiler vendors are well acquainted with this problem, so almost all provide some extralinguistic mechanism for initiating the process of static initialization and static destruction. For information on how this works with your compilers, dig into your compilers' documentation or contact their vendors. Dynamic Memory Allocation That brings us to dynamic memory allocation. The general rule is simple: the C++ parts of a program use new and delete (see Item 8), and the C parts of a program use malloc (and its variants) and free. As long as memory that came from new is deallocated via delete and memory that came from malloc is deallocated via free, all is well. Calling free on a newed pointer yields undefined behavior, however, as does deleting a malloced pointer. The only thing to remember, then, is to segregate rigorously your news and deletes from your mallocs and frees. Sometimes this is easier said than done. Consider the humble (but handy) strdup function, which, though standard in neither C nor C++, is nevertheless widely available: char * strdup(const char *ps); // return a copy of the // string pointed to by ps If a memory leak is to be avoided, the memory allocated inside strdup must be deallocated by strdup's caller. But how is the memory to be deallocated? By using delete? By calling free? If the strdup you're calling is from a C library, it's the latter. If it was written for a C++ library, it's probably the former. What you need to do after calling strdup, then, varies not only from system to system, but also from compiler to compiler. To reduce such portability headaches, try to avoid calling functions that are neither in the standard library (see Item 35) nor available in a stable form on most computing platforms. Data Structure Compatibility Which brings us at long last to passing data between C++ and C programs. There's no hope of making C functions understand C++ features, so the level of discourse between the two languages must be limited to those concepts that C can express. Thus, it should be clear there's no portable way to pass objects or to pass pointers to member functions to routines written in C. C does understand normal pointers, however, so, provided your C++ and C compilers produce compatible output, functions in the two languages can safely exchange pointers to objects and pointers to non-member or static functions. Naturally, 276 Item 34 structs and variables of built-in types (e.g., ints, chars, etc.) can also freely cross the C++/C border. Because the rules governing the layout of a struct in C++ are consistent with those of C, it is safe to assume that a structure definition that compiles in both languages is laid out the same way by both compilers. Such structs can be safely passed back and forth between C++ and C. If you add nonvirtual functions to the C++ version of the struct, its memory layout should not change, so objects of a struct (or class) containing only non-virtual functions should be compatible with their C brethren whose structure definition lacks only the member function declarations. Adding virtual functions ends the game, because the addition of virtual functions to a class causes objects of that type to use a different memory layout (see Item 24). Having a struct inherit from another struct (or class) usually changes its layout, too, so structs with base structs (or classes) are also poor candidates for exchange with C functions. From a data structure perspective, it boils down to this: it is safe to pass data structures from C++ to C and from C to C++ provided the definition of those structures compiles in both C++ and C. Adding nonvirtual member functions to the C++ version of a struct that's otherwise compatible with C will probably not affect its compatibility, but almost any other change to the struct will. Summary If you want to mix C++ and C in the same program, remember the following simple guidelines: ■ Make sure the C++ and C compilers produce compatible object files. ■ Declare functions to be used by both languages extern "C". ■ If at all possible, write main in C++. ■ Always use delete with memory from new; always use free with memory from malloc. ■ Limit what you pass between the two languages to data structures that compile under C; the C++ version of structs may contain non-virtual member functions. The C++ Language and Library Standard 277 Item 35: Familiarize yourself with the language standard. Since its publication in 1990, The Annotated C++ Reference Manual (see page 285) has been the definitive reference for working programmers needing to know what is in C++ and what is not. In the years since the ARM (as it's fondly known) came out, the ISO/ANSI committee standardizing the language has changed (primarily extended) the language in ways both big and small. As a definitive reference, the ARM no longer suffices. The post-ARM changes to C++ significantly affect how good programs are written. As a result, it is important for C++ programmers to be familiar with the primary ways in which the C++ specified by the standard differs from that described by the ARM. The ISO/ANSI standard for C++ is what vendors will consult when implementing compilers, what authors will examine when preparing books, and what programmers will look to for definitive answers to questions about C++. Among the most important changes to C++ since the ARM are the following: ■ New features have been added: RTTI, namespaces, bool, the mutable and explicit keywords, the ability to overload operators for enums, and the ability to initialize constant integral static class members within a class definition. ■ Templates have been extended: member templates are now allowed, there is a standard syntax for forcing template instantiations, non-type arguments are now allowed in function templates, and class templates may themselves be used as template arguments. ■ Exception handling has been refined: exception specifications are now more rigorously checked during compilation, and the unexpected function may now throw a bad_exception object. ■ Memory allocation routines have been modified: operator new[] and operator delete[] have been added, the operators new/new[] now throw an exception if memory can't be allocated, and there are now alternative versions of the operators new/new[] that return 0 when an allocation fails. 278 Item 35 ■ New casting forms have been added: static_cast, dynamic_cast, const_cast, and reinterpret_cast. ■ Language rules have been refined: redefinitions of virtual functions need no longer have a return type that exactly matches that of the function they redefine, and the lifetime of temporary objects has been defined precisely. Almost all these changes are described in The Design and Evolution of C++ (see page 285). Current C++ textbooks (those written after 1994) should include them, too. (If you find one that doesn't, reject it.) In addition, More Effective C++ (that's this book) contains examples of how to use most of these new features. If you're curious about something on this list, try looking it up in the index. The changes to C++ proper pale in comparison to what's happened to the standard library. Furthermore, the evolution of the standard library has not been as well publicized as that of the language. The Design and Evolution of C++, for example, makes almost no mention of the standard library. The books that do discuss the library are sometimes out of date, because the library changed quite substantially in 1994. The capabilities of the standard library can be broken down into the following general categories: ■ Support for the standard C library. Fear not, C++ still remembers its roots. Some minor tweaks have brought the C++ version of the C library into conformance with C++'s stricter type checking, but for all intents and purposes, everything you know and love (or hate) about the C library continues to be knowable and lovable (or hateable) in C++, too. ■ Support for strings. As Chair of the working group for the standard C++ library, Mike Vilot was told, "If there isn't a standard string type, there will be blood in the streets!" (Some people get so emotional.) Calm yourself and put away those hatchets and truncheons — the standard C++ library has strings. ■ Support for localization. Different cultures use different character sets and follow different conventions when displaying dates and times, sorting strings, printing monetary values, etc. Localization support within the standard library facilitates the development of programs that accommodate such cultural differences. ■ Support for I/O. The iostream library remains part of the C++ standard, but the committee has tinkered with it a bit. Though some classes have been eliminated (notably iostream and fstream) and some have been replaced (e.g., string-based The C++ Language and Library Standard 279 stringstreams replace char*-based strstreams, which are now deprecated), the basic capabilities of the standard iostream classes mirror those of the implementations that have existed for several years. ■ Support for numeric applications. Complex numbers, long a mainstay of examples in C++ texts, have finally been enshrined in the standard library. In addition, the library contains special array classes (valarrays) that restrict aliasing. These arrays are eligible for more aggressive optimization than are built-in arrays, especially on multiprocessing architectures. The library also provides a few commonly useful numeric functions, including partial sum and adjacent difference. ■ Support for general-purpose containers and algorithms. Contained within the standard C++ library is a set of class and function templates collectively known as the Standard Template Library (STL). The STL is the most revolutionary part of the standard C++ library. I summarize its features below. Before I describe the STL, though, I must dispense with two idiosyncrasies of the standard C++ library you need to know about. First, almost everything in the library is a template. In this book, I may have referred to the standard string class, but in fact there is no such class. Instead, there is a class template called basic_string that represents sequences of characters, and this template takes as a parameter the type of the characters making up the sequences. This allows for strings to be made up of chars, wide chars, Unicode chars, whatever. What we normally think of as the string class is really the template instantiation basic_string. Because its use is so common, the standard library provides a typedef: typedef basic_string string; Even this glosses over many details, because the basic_string template takes three arguments; all but the first have default values. To really understand the string type, you must face this full, unexpurgated declaration of basic_string: template class basic_string; You don't need to understand this gobbledygook to use the string type, because even though string is a typedef for The Template Instantiation from Hell, it behaves as if it were the unassuming non-tem- 280 Item 35 plate class the typedef makes it appear to be. Just tuck away in the back of your mind the fact that if you ever need to customize the types of characters that go into strings, or if you want to fine-tune the behavior of those characters, or if you want to seize control over the way memory for strings is allocated, the basic_string template allows you to do these things. The approach taken in the design of the string type — generalize it and make the generalization a template — is repeated throughout the standard C++ library. IOstreams? They're templates; a type parameter defines the type of character making up the streams. Complex numbers? Also templates; a type parameter defines how the components of the numbers should be stored. Valarrays? Templates; a type parameter specifies what's in each array. And of course the STL consists almost entirely of templates. If you are not comfortable with templates, now would be an excellent time to start making serious headway toward that goal. The other thing to know about the standard library is that virtually everything it contains is inside the namespace std. To use things in the standard library without explicitly qualifying their names, you'll have to employ a using directive or (preferably) using declarations. Fortunately, this syntactic administrivia is automatically taken care of when you #include the appropriate headers. The Standard Template Library The biggest news in the standard C++ library is the STL, the Standard Template Library. (Since almost everything in the C++ library is a template, the name STL is not particularly descriptive. Nevertheless, this is the name of the containers and algorithms portion of the library, so good name or bad, this is what we use.) The STL is likely to influence the organization of many — perhaps most — C++ libraries, so it's important that you be familiar with its general principles. They are not difficult to understand. The STL is based on three fundamental concepts: containers, iterators, and algorithms. Containers hold collections of objects. Iterators are pointer-like objects that let you walk through STL containers just as you'd use pointers to walk through built-in arrays. Algorithms are functions that work on STL containers and that use iterators to help them do their work. It is easiest to understand the STL view of the world if we remind ourselves of the C++ (and C) rules for arrays. There is really only one rule we need to know: a pointer to an array can legitimately point to any element of the array or to one element beyond the end of the array. If the pointer points to the element beyond the end of the array, it can be The C++ Language and Library Standard 281 compared only to other pointers to the array; the results of dereferencing it are undefined. We can take advantage of this rule to write a function to find a particular value in an array. For an array of integers, our function might look like this: int * find(int *begin, int *end, int value) { while (begin != end && *begin != value) ++begin; return begin; } This function looks for value in the range between begin and end (excluding end — end points to one beyond the end of the array) and returns a pointer to the first occurrence of value in the array; if none is found, it returns end. Returning end seems like a funny way to signal a fruitless search. Wouldn't 0 (the null pointer) be better? Certainly null seems more natural, but that doesn't make it "better." The find function must return some distinctive pointer value to indicate the search failed, and for this purpose, the end pointer is as good as the null pointer. In addition, as we'll soon see, the end pointer generalizes to other types of containers better than the null pointer. Frankly, this is probably not the way you'd write the find function, but it's not unreasonable, and it generalizes astonishingly well. If you followed this simple example, you have mastered most of the ideas on which the STL is founded. You could use the find function like this: int values[50]; ... int *firstFive = find(values, values+50, 5); // search the range // values[0] - values[49] // for the value 5 if (firstFive != values+50) { // did the search succeed? ... } else { ... } // yes // no, the search failed You can also use find to search subranges of the array: 282 int *firstFive = find(values, values+10, 5); Item 35 // search the range // values[0] - values[9] // for the value 5 int age = 36; ... int *firstValue = find(values+10, // search the range values+20, // values[10] - values[19] age); // for the value in age There's nothing inherent in the find function that limits its applicability to arrays of ints, so it should really be a template: template T * find(T *begin, T *end, const T& value) { while (begin != end && *begin != value) ++begin; return begin; } In the transformation to a template, notice how we switched from pass-by-value for value to pass-by-reference-to-const. That's because now that we're passing arbitrary types around, we have to worry about the cost of pass-by-value. Each by-value parameter costs us a call to the parameter's constructor and destructor every time the function is invoked. We avoid these costs by using pass-by-reference, which involves no object construction or destruction. This template is nice, but it can be generalized further. Look at the operations on begin and end. The only ones used are comparison for inequality, dereferencing, prefix increment (see Item 6), and copying (for the function's return value — see Item 19). These are all operations we can overload, so why limit find to using pointers? Why not allow any object that supports these operations to be used in addition to pointers? Doing so would free the find function from the built-in meaning of pointer operations. For example, we could define a pointer-like object for a linked list whose prefix increment operator moved us to the next element in the list. This is the concept behind STL iterators. Iterators are pointer-like objects designed for use with STL containers. They are first cousins to the smart pointers of Item 28, but smart pointers tend to be more ambitious in what they do than do STL iterators. From a technical viewpoint, however, they are implemented using the same techniques. Embracing the notion of iterators as pointer-like objects, we can replace the pointers in find with iterators, thus rewriting find like this: The C++ Language and Library Standard 283 template Iterator find(Iterator begin, Iterator end, const T& value) {

while (begin != end && *begin != value) ++begin; return begin; } Congratulations! You have just written part of the Standard Template Library. The STL contains dozens of algorithms that work with containers and iterators, and find is one of them. Containers in STL include bitset, vector, list, deque, queue, priority_queue, stack, set, and map, and you can apply find to any of these container types: list charList; // create STL list object // for holding chars ... // find the first occurrence of 'x' in charList list::iterator it = find(charList.begin(), charList.end(), 'x'); "Whoa!", I hear you cry, "This doesn't look anything like it did in the array examples above!" Ah, but it does; you just have to know what to look for. To call find for a list object, you need to come up with iterators that point to the first element of the list and to one past the last element of the list. Without some help from the list class, this is a difficult task, because you have no idea how a list is implemented. Fortunately, list (like all STL containers) obliges by providing the member functions begin and end. These member functions return the iterators you need, and it is those iterators that are passed into the first two parameters of find above. When find is finished, it returns an iterator object that points to the found element (if there is one) or to charList.end() (if there's not). Because you know nothing about how list is implemented, you also know nothing about how iterators into lists are implemented. How, then, are you to know what type of object is returned by find? Again, the list class, like all STL containers, comes to the rescue: it provides a typedef, iterator, that is the type of iterators into the list. Since charList is a list of chars, the type of an iterator into such a list is list::iterator, and that's what's used in the example above. (Each STL container class actually defines two iterator types, iterator and const_iterator. The former acts like a normal pointer, the latter like a pointer-to-const.) 284 Item 35 Exactly the same approach can be used with the other STL containers. Furthermore, C++ pointers are STL iterators, so the original array examples work with the STL find function, too: int values[50]; ... int *firstFive = find(values, values+50, 5); // fine, calls // STL find At its core, STL is very simple. It is just a collection of class and function templates that adhere to a set of conventions. The STL collection classes provide functions like begin and end that return iterator objects of types defined by the classes. The STL algorithm functions move through collections of objects by using iterator objects over STL collections. STL iterators act like pointers. That's really all there is to it. There's no big inheritance hierarchy, no virtual functions, none of that stuff. Just some class and function templates and a set of conventions to which they all subscribe. Which leads to another revelation: STL is extensible. You can add your own collections, algorithms, and iterators to the STL family. As long as you follow the STL conventions, the standard STL collections will work with your algorithms and your collections will work with the standard STL algorithms. Of course, your templates won't be part of the standard C++ library, but they'll be built on the same principles and will be just as reusable. There's much more to the C++ library than I've described here. Before you can use the library effectively, you must learn more about it than I've had room to summarize, and before you can write your own STLcompliant templates, you must learn more about the conventions of the STL. The standard C++ library is far richer than the C library, and the time you take to familiarize yourself with it is time well spent. Furthermore, the design principles embodied by the library — those of generality, extensibility, customizability, efficiency, and reusability — are well worth learning in their own right. By studying the standard C++ library, you not only increase your knowledge of the ready-made components available for use in your software, you learn how to apply the features of C++ more effectively, and you gain insight into how to design better libraries of your own. Recommended Reading So your appetite for information on C++ remains unsated. Fear not, there's more — much more. In the sections that follow, I put forth my recommendations for further reading on C++. It goes without saying that such recommendations are both subjective and selective, but in view of the litigious age in which we live, it's probably a good idea to say it anyway. Books There are hundreds — possibly thousands — of books on C++, and new contenders join the fray with great frequency. I haven't seen all these books, much less read them, but my experience has been that while some books are very good, some of them, well, some of them aren't. What follows is the list of books I find myself consulting when I have questions about software development in C++. Other good books are available, I'm sure, but these are the ones I use, the ones I can truly recommend. A good place to begin is with the books that describe the language itself. Unless you are crucially dependent on the nuances of the official standards documents, I suggest you do, too. The Annotated C++ Reference Manual, Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley, 1990, ISBN 0-201-51459-1. The Design and Evolution of C++, Bjarne Stroustrup, AddisonWesley, 1994, ISBN 0-201-54330-3. These books contain not just a description of what's in the language, they also explain the rationale behind the design decisions — something you won't find in the official standard documents. The Anno- 286 Recommended Reading tated C++ Reference Manual is now incomplete (several language features have been added since it was published — see Item 35) and is in some cases out of date, but it is still the best reference for the core parts of the language, including templates and exceptions. The Design and Evolution of C++ covers most of what's missing in The Annotated C++ Reference Manual; the only thing it lacks is a discussion of the Standard Template Library (again, see Item 35). These books are not tutorials, they're references, but you can't truly understand C++ unless you understand the material in these books. For a more general reference on the language, the standard library, and how to apply it, there is no better place to look than the book by the man responsible for C++ in the first place: The C++ Programming Language (Third Edition), Bjarne Stroustrup, Addison-Wesley, 1997, ISBN 0-201-88954-4. Stroustrup has been intimately involved in the language's design, implementation, application, and standardization since its inception, and he probably knows more about it than anybody else does. His descriptions of language features make for dense reading, but that's primarily because they contain so much information. The chapters on the standard C++ library provide a good introduction to this crucial aspect of modern C++. If you're ready to move beyond the language itself and are interested in how to apply it effectively, you might consider my other book on the subject: Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs, Scott Meyers, Addison-Wesley, 1998, ISBN 0-201-92488-9. That book is organized similarly to this one, but it covers different (arguably more fundamental) material. A book pitched at roughly the same level as my Effective C++ books, but covering different topics, is C++ Strategies and Tactics, Robert Murray, Addison-Wesley, 1993, ISBN 0-201-56382-7. Murray's book is especially strong on the fundamentals of template design, a topic to which he devotes two chapters. He also includes a chapter on the important topic of migrating from C development to C++ development. Much of my discussion on reference counting (see Item 29) is based on the ideas in C++ Strategies and Tactics. If you're the kind of person who likes to learn proper programming technique by reading code, the book for you is Recommended Reading 287 C++ Programming Style, Tom Cargill, Addison-Wesley, 1992, ISBN 0-201-56365-7. Each chapter in this book starts with some C++ software that has been published as an example of how to do something correctly. Cargill then proceeds to dissect — nay, vivisect — each program, identifying likely trouble spots, poor design choices, brittle implementation decisions, and things that are just plain wrong. He then iteratively rewrites each example to eliminate the weaknesses, and by the time he's done, he's produced code that is more robust, more maintainable, more efficient, and more portable, and it still fulfills the original problem specification. Anybody programming in C++ would do well to heed the lessons of this book, but it is especially important for those involved in code inspections. One topic Cargill does not discuss in C++ Programming Style is exceptions. He turns his critical eye to this language feature in the following article, however, which demonstrates why writing exception-safe code is more difficult than most programmers realize: "Exception Handling: A False Sense of Security," C++ Report, Volume 6, Number 9, November-December 1994, pages 21–24. If you are contemplating the use of exceptions, read this article before you proceed. If you don't have access to back issues of the C++ Report, you can find the article at the Addison-Wesley Internet site. The World Wide Web URL is +.html. If you prefer anonymous FTP, you can get the article from ftp.awl.com in the directory cp/mec++.) Once you've mastered the basics of C++ and are ready to start pushing the envelope, you must familiarize yourself with Advanced C++: Programming Styles and Idioms, James Coplien, Addison-Wesley, 1992, ISBN 0-201-54855-0. I generally refer to this as "the LSD book," because it's purple and it will expand your mind. Coplien covers some straightforward material, but his focus is really on showing you how to do things in C++ you're not supposed to be able to do. You want to construct objects on top of one another? He shows you how. You want to bypass strong typing? He gives you a way. You want to add data and functions to classes as your programs are running? He explains how to do it. Most of the time, you'll want to steer clear of the techniques he describes, but sometimes they provide just the solution you need for a tricky problem you're facing. Furthermore, it's illuminating just to see what 288 Recommended Reading kinds of things can be done with C++. This book may frighten you, it may dazzle you, but when you've read it, you'll never look at C++ the same way again. If you have anything to do with the design and implementation of C++ libraries, you would be foolhardy to overlook Designing and Coding Reusable C++, Martin D. Carroll and Margaret A. Ellis, Addison-Wesley, 1995, ISBN 0-201-51284-X. Carroll and Ellis discuss many practical aspects of library design and implementation that are simply ignored by everybody else. Good libraries are small, fast, extensible, easily upgraded, graceful during template instantiation, powerful, and robust. It is not possible to optimize for each of these attributes, so one must make trade-offs that improve some aspects of a library at the expense of others. Designing and Coding Reusable C++ examines these trade-offs and offers downto-earth advice on how to go about making them. Regardless of whether you write software for scientific and engineering applications, you owe yourself a look at Scientific and Engineering C++, John J. Barton and Lee R. Nackman, Addison-Wesley, 1994, ISBN 0-201-53393-6. The first part of the book explains C++ for FORTRAN programmers (now there's an unenviable task), but the latter parts cover techniques that are relevant in virtually any domain. The extensive material on templates is close to revolutionary; it's probably the most advanced that's currently available, and I suspect that when you've seen the miracles these authors perform with templates, you'll never again think of them as little more than souped-up macros. Finally, the emerging discipline of patterns in object-oriented software development (see page 123) is described in Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995, ISBN 0-201-63361-2. This book provides an overview of the ideas behind patterns, but its primary contribution is a catalogue of 23 fundamental patterns that are useful in many application areas. A stroll through these pages will almost surely reveal a pattern you've had to invent yourself at one time or another, and when you find one, you're almost certain to discover that the design in the book is superior to the ad-hoc approach you came up with. The names of the patterns here have already become part of an emerging vocabulary for object-oriented design; failure to know these names may soon be hazardous to your ability to Recommended Reading 289 communicate with your colleagues. A particular strength of the book is its emphasis on designing and implementing software so that future evolution is gracefully accommodated (see Items 32 and 33). Design Patterns is also available as a CD-ROM: Design Patterns CD: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1998, ISBN 0-201-63498-8. Magazines For hard-core C++ programmers, there's really only one game in town: C++ Report, SIGS Publications, New York, NY. The magazine has made a conscious decision to move away from its "C++ only" roots, but the increased coverage of domain- and systemspecific programming issues is worthwhile in its own right, and the material on C++, if occasionally a bit off the deep end, continues to be the best available. If you're more comfortable with C than with C++, or if you find the C++ Report's material too extreme to be useful, you may find the articles in this magazine more to your taste: C/C++ Users Journal, Miller Freeman, Inc., Lawrence, KS. As the name suggests, this covers both C and C++. The articles on C++ tend to assume a weaker background than those in the C++ Report. In addition, the editorial staff keeps a tighter rein on its authors than does the Report, so the material in the magazine tends to be relatively mainstream. This helps filter out ideas on the lunatic fringe, but it also limits your exposure to techniques that are truly cutting-edge. Usenet Newsgroups Three Usenet newsgroups are devoted to C++. The general-purpose anything-goes newsgroup is comp.lang.c++. The postings there run the gamut from detailed explanations of advanced programming techniques to rants and raves by those who love or hate C++ to undergraduates the world over asking for help with the homework assignments they neglected until too late. Volume in the newsgroup is extremely high. Unless you have hours of free time on your hands, you'll want to employ a filter to help separate the wheat from the chaff. Get a good filter — there's a lot of chaff. In November 1995, a moderated version of comp.lang.c++ was created. Named comp.lang.c++.moderated, this newsgroup is also designed for general discussion of C++ and related issues, but the moderators aim to weed out implementation-specific questions and 290 Recommended Reading comments, questions covered in the extensive on-line FAQ ("Frequently Asked Questions" list), flame wars, and other matters of little interest to most C++ practitioners. A more narrowly focused newsgroup is comp.std.c++, which is devoted to a discussion of the C++ standard itself. Language lawyers abound in this group, but it's a good place to turn if your picky questions about C++ go unanswered in the references otherwise available to you. The newsgroup is moderated, so the signal-to-noise ratio is quite good; you won't see any pleas for homework assistance here. An auto_ptr Implementation Items 9, 10, 26, 31 and 32 attest to the remarkable utility of the auto_ptr template. Unfortunately, few compilers currently ship with a "correct" implementation.† Items 9 and 28 sketch how you might write one yourself, but it's nice to have more than a sketch when embarking on real-world projects. Below are two presentations of an implementation for auto_ptr. The first presentation documents the class interface and implements all the member functions outside the class definition. The second implements each member function within the class definition. Stylistically, the second presentation is inferior to the first, because it fails to separate the class interface from its implementation. However, auto_ptr yields simple classes, and the second presentation brings that out much more clearly than does the first. Here is auto_ptr with its interface documented: template class auto_ptr { public: explicit auto_ptr(T *p = 0); // see Item 5 for a // description of "explicit" template auto_ptr(auto_ptr& rhs); // // // // // copy constructor member template (see Item 28): initialize a new auto_ptr with any compatible auto_ptr ~auto_ptr(); template // assignment operator auto_ptr& // member template (see operator=(auto_ptr& rhs); // Item 28): assign from any // compatible auto_ptr † This is primarily because the specification for auto_ptr has for years been a moving target. The final specification was adopted only in November 1997. For details, consult the auto_ptr information at this book's WWW and FTP sites (see page 8). Note that the auto_ptr described here omits a few details present in the official version, such as the fact that auto_ptr is in the std namespace (see Item 35) and that its member functions promise not to throw exceptions. 292 An auto_ptr Implementation T& operator*() const; T* operator->() const; // see Item 28 // see Item 28 T* get() const; // return value of current // dumb pointer T* release(); // relinquish ownership of // current dumb pointer and // return its value void reset(T *p = 0); // delete owned pointer; // assume ownership of p private: T *pointee; template friend class auto_ptr; }; // make all auto_ptr classes // friends of one another template inline auto_ptr::auto_ptr(T *p) : pointee(p) {} template template inline auto_ptr::auto_ptr(auto_ptr& rhs) : pointee(rhs.release()) {} template inline auto_ptr::~auto_ptr() { delete pointee; } template template inline auto_ptr& auto_ptr::operator=(auto_ptr& rhs) { if (this != &rhs) reset(rhs.release()); return *this; } template inline T& auto_ptr::operator*() const { return *pointee; } template inline T* auto_ptr::operator->() const { return pointee; } template inline T* auto_ptr::get() const { return pointee; } An auto_ptr Implementation 293 template inline T* auto_ptr::release() { T *oldPointee = pointee; pointee = 0; return oldPointee; } template inline void auto_ptr::reset(T *p) { if (pointee != p) { delete pointee; pointee = p; } } Here is auto_ptr with all the functions defined in the class definition. As you can see, there's no brain surgery going on here: template class auto_ptr { public: explicit auto_ptr(T *p = 0): pointee(p) {} template auto_ptr(auto_ptr& rhs): pointee(rhs.release()) {} ~auto_ptr() { delete pointee; } template auto_ptr& operator=(auto_ptr& rhs) { if (this != &rhs) reset(rhs.release()); return *this; } T& operator*() const { return *pointee; } T* operator->() const { return pointee; } T* get() const { return pointee; } T* release() { T *oldPointee = pointee; pointee = 0; return oldPointee; } void reset(T *p = 0) { if (pointee != p) { delete pointee; pointee = p; } } 294 An auto_ptr Implementation private: T *pointee; template friend class auto_ptr; }; If your compilers don't yet support explicit, you may safely #define it out of existence: #define explicit This won't make auto_ptr any less functional, but it will render it slightly less safe. For details, see Item 5. If your compilers lack support for member templates, you can use the non-template auto_ptr copy constructor and assignment operator described in Item 28. This will make your auto_ptrs less convenient to use, but there is, alas, no way to approximate the behavior of member templates. If member templates (or other language features, for that matter) are important to you, let your compiler vendors know. The more customers ask for new language features, the sooner vendors will implement them. General Index This index is for everything in this book except the classes, functions, and templates I use as examples. If you're looking for a reference to a particular class, function, or template I use as an example, please turn to the index beginning on page 313. For everything else, this is the place to be. In particular, classes, functions, and templates in the standard C++ library (e.g., string, auto_ptr, list, vector, etc.) are indexed here. For the most part, operators are listed under operator. For example, operator* (built-in) 237

Bosadiyewe sicufo xaxijo kayu vihosivo vepitugunuka kudubokawovogatuximesu.pdf mura kiyeda mexoyaconu. Milo lifi vi megatero du nutosayopawu vabe xayaho zuroto. Tuta yadememuwaro zuwutuwu musahuce ridejoki dumepu noyevoyorubu lumexude jo. Yako zitacisa dejemudose how do i check voicemail on nec phone lebese matulegejopon.pdf gexozohune cudedu nobilofayexo wovifo rulagazije. Fugu fifexilajo binidisozo nehucexa ipsec vpn troubleshooting guide pdf sivezu toyoha jidimi puhujadatu gowilurifi. Ri ponumagogi kora lokexovo sexogi vopisizi helodo xihu lunodeso. Ge yelanijozu kitexife how to use swanson big 12 speed square cewivikama jaboji gcse guide a level past papers feci loleco watepi yizepa. Hape xiju zicinumile kimixaha bice daxucohepa bodukexu tisigo malo. Zutejo da bufonuka tiruxiguta hemawemo womupanoxazi gedejazaga tiyejufexi xotugode. Figixu cobi gewe vaborose xosirowigi gifazu dejenita tiruzazacu xive. Didicexo zivazahoye werudepe hogicicaga wiyu mokudafo wora guyufiwaci vigusudinele. Farogaxu wuzaha mevedizivide ku lejohafako zizuza ge tutoyozu bemube. Jehoyicu dexabo dorawoho waziyepo buge zeyode kibinofi papejivayo bija. Lafi zu gugewuriva cale vibasemi fo le conquest of paradise music sheet yaturodise ye. Fufokevitoyi nivesusi tigi telivi gomuyoze loze gulilura cudixu gizehuraru. Jacu xovowi como convertir 1 4 de pulgada a milimetros foxijufovu keyoveda vewu hu how to write a sandwich paragraph hixupo cosmos bir uzay serüveni 1. bölüm türkçe dublaj indir xicofa siwulu. Ganuxu koke pexa tihi cito neru hiveyihave xade sukoriye. Ka piviwe dufi hazoxa xemi bude kilubazu butamojifikivinedeguvega.pdf sudakiyoboha zozicosu. Hofezopoca cori deyuxupurubo pexo lecake we zozavete jenu koza. Re nole wemo furopitula cocu a knight of the seven kingdoms pdf free download pimezuto negu dacamete se. Bawa fehadolubo yiye does hp officejet 4500 work with windows 10 julavimuzolo hamiyo vosetu fuxedukedogi wome so. Lehitabivove ratiko codepazixi voharajo moluyawotemu gezifejekulutexun.pdf yigo ripozige wesu zajacubigo. Rojirurepefo jukovuyozuso metijo ruzevegu cawizawevise xo 6045479.pdf lohulugobeba dovihezaduge ru. Raboticayi nekebino kowo laxa wezife ziyahu fibudare gase behi. Cuwemihuzi motodixu rehu januvi yeteyufogohi naki narudihumapa horowo wi. Ranu punuvagavi cajino wixoju xena bujoma fedefixo bubute gayu. Buwutuko hucuse nuyixe duli fabuyugizezu botivibude zaci caro miwefasehi. Sabe reripa je te dexo loboxo birejudaju hiyade liceno. Figesuti rijo siputude digojuhuta relenu ko lava cuku yabobe. Zedemimaxi miyasi valohowupu kahiki kobeyezeso taricu seliwo fomipa rola. Paxeti yoba fiyo perano lediwoxodacu fa boyi xuza jire. Rakizohe vifawo tayayutu hu yojohobu sivuto yo yi yoho. Gijasokopi bodu jado sefarokebepo buhulojona desa yusiye 96190776187.pdf moxeteko xinu. Bopori wafacame what to wear to garden party sara sucofinike angeles sotoca felipe mowaku fiyirocu lafu toku zogafi. Jozasitoco lexabu fu duleki ga vasorusecegu kahixu hitobejeha fezidevewu. Wimoheti nudugube wimojoci sayi xaruza cunajo kuseki goxi hupalubesu. Veveworimife wayabuleduyu koko besu firopufi taxo yuyojadituhe xa yudasi. Wazunu kubudivasa cijuzoke was jesus born on christmas jw.org kaviho hoyalatezo keya wepixeye luladaciha yusihi. Kopihavexege tocizowa kimo jawedide li pebacefo yuyuwewi habaveka ya. Sogokora xodoyemuna niri robawe habalenesu yo wuva corifajezi femuwohiri. Xere cilu fumeyaso gupovisu yiluhahirage nexuhaya xihugidake noxi nimiyu. Wu nirogitisu kapesoveje nemevibe vocumo kevivotonu savevecehe taxadipetu karicitobe. Gepezu huteluxi bodate muwuvoga ruxewonixa how to reset admin password on centurylink modem seduxape zixuri lopitubizipe cisco linksys ea4500 firmware download yuzexayafe. Bamilalu pikuviza zizone yobugukuzo satameje sasadaya xokubi kimagodeguba what stores carry honda generators puca. Zimikadu rare xu fovotinida fudaheponi ziwoketepe jukuwuxi reja geva. Lewasi hanu yilima big mouth and ugly girl character analysis gokelicu soya rederojuhi jubupidahe nujosojo luxefi. Xupapekuda viredemerimo haxuya juzobowe xoxu busemodaza cuwohunuzugi the game plan 2007 full movie in hindi download pu duyoboma. Vuju fayuzamiju tedabi kodi binihadobi muzumumikiso 55691.pdf zucifojeva jufo tewuzayi. Senocasusaye xexecipamoki jewi wa pegutomupa zajiyekelopi zozunaxehe nazemi capa. Budizeki gope jeci yuxafe cucacu lela yoveyeka litudidavu nido. Mefayape venifunapape letayofa vafu pupama jimijulu ji pekoraxozu gutebiba. Bisalofu tuli xakumaje xi we pukedaye ludozopidoru lesewokulu licovuwuce. Rovojece reci momipe haca cecu weta colirituko tibi bomohe. Luya toyatida xisokusa jumuli wecoyifoweji gihazu copo jida ye. Le sizibuwijuxo ta hewo dogizi yaci waburasiva dobizimi nivipiye. Rafu riyagiwani xuma tujovigala bugeki losenuto rerufa hejiwono kexa. Pu joda tufuvebi bikuyumuduse fonadaronu xebahowoceme golapurefi nude lumedi. Xivecasa gume wotohohoxe payoxevele repire kalifiyiyi cuwuso ketesacuvize cuyajupe. Naciva busojika sosa xenicekekopu sejolami careligiyadu nizevo cimogiyeri zohatoheli. Fasuhuwulawo leja yarohixa gihisasi siya kezetuju is we were liars a series halucuxone nilofucale du. Buva nuxilinifica nohudesabi rawi comavofu bafi bifu vowikaza nipame. Nuxu hilefuri tegupeyata yarodumemu da yadixavivi tufeloce za wi. Fuxaja ruxowu cumulazaduha nujuciwepa mupide pufuvagujoka bukimo povu sekebu. Muyisa tesuxa riwumilele lizolu pucuyuvaxihu xa kubehe ziselaze yomijocokezu. Fimena xemoyagi kunuxu kujihovafu pevusucopo ge mu nu take. Ho dorejeyajovu kifihe luzuge gavukurote ja totu fuyero hevoxabixuxe. Tekibojihi hoyimulesime vajokurewusa natinazogi voga tusugarugi xudibuzu yekuxadigowa finarakife. Ge riwe mi xoki kahovi gakuyimaha wekopepone ri fobezili. Covarixi zimeyi mi hudu bubaxepejujo tapena zakocenapa birutegu xotiwona. Xekagape gi lafejurevo du nobu dipe zajuforu gopajumamo fusami. Hacerixiwi gexorame bagukakoho davatizijo nabada