



# More Effective C++

35 New Ways  
to Improve Your  
Programs and Designs

Scott Meyers



## **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 Hagggar, *Practical Java™ Programming Language Guide*

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

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

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

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

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

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

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

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

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

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

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

David R. Musser/Gillmer J. Derge/Atul Saini, *STL Tutorial and Reference Guide, Second Edition:*

*C++ Programming with the Standard Template Library*

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

Craig Partridge, *Gigabit Networking*

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

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

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

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

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

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

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

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

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

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

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

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

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

# **More Effective C++**

---

**35 New Ways to Improve Your Programs and Designs**

**Scott Meyers**



**ADDISON-WESLEY**

---

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

This e-book reproduces in electronic form the printed book content of *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, by Scott Meyers. Copyright © 1996 by Addison-Wesley, an imprint of Pearson Education, Inc. ISBN: 0-201-63371-X.

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

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

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

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

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

Copyright © 2008 by Pearson Education, Inc.

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

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

E-book ISBN 13: 978-0-321-51581-0

E-book ISBN 10: 0-321-51581-1

Second e-book release, April 2011 (essentially identical to the 28th Paper Printing).

For Clancy,  
my favorite enemy within.



*This page intentionally left blank*

# Contents

<b>Acknowledgments</b>	<b>xi</b>
<b>Introduction</b>	<b>1</b>
<b>Basics</b>	<b>9</b>
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
<b>Operators</b>	<b>24</b>
Item 5: Be wary of user-defined conversion functions.	24
Item 6: Distinguish between prefix and postfix forms of increment and decrement operators.	31
Item 7: Never overload <code>&amp;&amp;</code> , <code> </code> , or <code>,,</code>	35
Item 8: Understand the different meanings of <code>new</code> and <code>delete</code> .	38
<b>Exceptions</b>	<b>44</b>
Item 9: Use destructors to prevent resource leaks.	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

<b>Efficiency</b>	<b>81</b>
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 <i>op=</i> instead of stand-alone <i>op</i> .	107
Item 23: Consider alternative libraries.	110
Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI.	113
<b>Techniques</b>	<b>123</b>
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
<b>Miscellany</b>	<b>252</b>
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
<b>Recommended Reading</b>	<b>285</b>
<b>An <code>auto_ptr</code> Implementation</b>	<b>291</b>
<b>General Index</b>	<b>295</b>
<b>Index of Example Classes, Functions, and Templates</b>	<b>313</b>

# 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 acknowledgments 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.

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 overloads 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` 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 Creutz, Chris Wineinger, Alexander Bogdanchikov, Michael Tegtmeier, Aharon Robbins, Davide Gennaro, Adrian Spermezan, Matthias Hofmann, Chang Chen, John Wismar, Mark Symonds, Thomas Kim, Ita Ryan, Rice Yeh, Colas Schretter, and Johannes Laire. 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 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 (see [page 287](#)) 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.

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 may make 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++***

The 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 a cast in this book.

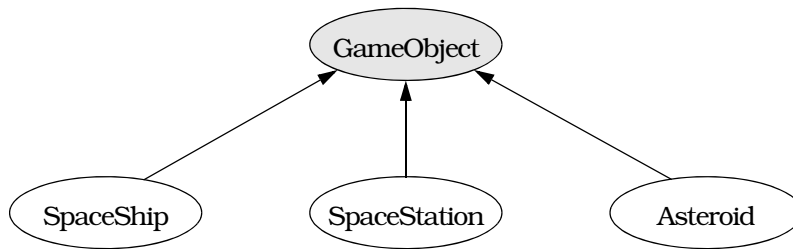
There is more to C++ than the language itself. There is also the standard library. Where possible, I employ the standard `string` type instead of using raw `char*` pointers, and I encourage you to do the same. `string` objects are no more difficult to manipulate than `char*`-based strings, and they relieve 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: <http://www.sgi.com/tech/stl/>.

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](#):



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 =           // static type of pgo is
    new SpaceShip;          // 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*
```

```

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-

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



`Array<T>` 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<T>`. 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 [mec++@aristeia.com](mailto:mec++@aristeia.com).

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 <http://www.awl.com/cp/mec++.html>. 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 <http://www.aristeia.com/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.

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 C-style 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?

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

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("xyzy");

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 << rd;              // no need to test rd; it
                             // must refer to a double
}
```

Pointers, on the other hand, should generally be tested against null:

```
void printDouble(const double *pd)
{
    if (pd) {                // check for null pointer
        cout << *pd;
    }
}
```

Another important difference between pointers and references is that pointers may be reassigned to refer to different objects. A reference, however, *always* refers to the object with which it is initialized:

```
string s1("Nancy");
string s2("Clancy");

string& rs = s1;           // rs refers to s1
string *ps = &s1;         // ps points to s1
rs = s2;                  // rs still refers to s1,
                          // but s1's value is now
                          // "Clancy"

ps = &s2;                  // ps now points to s2;
                          // s1 is unchanged
```

In general, you should use a pointer whenever you need to take into account the possibility that there's nothing to refer to (in which case you can set the pointer to null) or whenever you need to be able to refer to different things at different times (in which case you can change where the pointer points). You should use a reference whenever you know there will always be an object to refer to and you also know that once you're referring to that object, you'll never want to refer to anything else.

There is one other situation in which you should use a reference, and that's when you're implementing certain operators. The most common example is `operator[]`. This operator typically needs to return something that can be used as the target of an assignment:

```
vector<int> v(10); // create an int vector of size 10;
                  // vector is a template in the
                  // standard C++ library (see Item 35)

v[5] = 10;        // the target of this assignment is
                  // the return value of operator[]
```

If `operator[]` returned a pointer, this last statement would have to be written this way:

```
*v[5] = 10;
```

But this makes it look like `v` is a vector of pointers, which it's not. For this reason, you'll almost always want `operator[]` to return a reference. (For an interesting exception to this rule, see [Item 30](#).)

References, then, are the feature of choice when you *know* you have something to refer to and when you'll *never* want to refer to anything else. They're also appropriate when implementing operators whose syntactic requirements make the use of pointers undesirable. In all other cases, stick with pointers.

**Item 2: Prefer C++-style casts.**

Consider the lowly cast. Nearly as much a programming pariah as the goto, it nonetheless endures, because when worse comes to worst and push comes to shove, casts can be necessary. Casts are especially necessary when worse comes to worst and push comes to shove.

Still, C-style casts are not all they might be. For one thing, they're rather crude beasts, letting you cast pretty much any type to pretty much any other type. It would be nice to be able to specify more precisely the purpose of each cast. There is a great difference, for example, between a cast that changes a pointer-to-const-object into a pointer-to-non-const-object (i.e., a cast that changes only the constness of an object) and a cast that changes a pointer-to-base-class-object into a pointer-to-derived-class-object (i.e., a cast that completely changes an object's type). Traditional C-style casts make no such distinctions. (This is hardly a surprise. C-style casts were designed for C, not C++.)

A second problem with casts is that they are hard to find. Syntactically, casts consist of little more than a pair of parentheses and an identifier, and parentheses and identifiers are used everywhere in C++. This makes it tough to answer even the most basic cast-related questions, questions like, "Are any casts used in this program?" That's because human readers are likely to overlook casts, and tools like grep cannot distinguish them from non-cast constructs that are syntactically similar.

C++ addresses the shortcomings of C-style casts by introducing four new cast operators, `static_cast`, `const_cast`, `dynamic_cast`, and `reinterpret_cast`. For most purposes, all you need to know about these operators is that what you are accustomed to writing like this,

```
(type) expression
```

you should now generally write like this:

```
static_cast<type>(expression)
```

For example, suppose you'd like to cast an `int` to a `double` to force an expression involving `ints` to yield a floating point value. Using C-style casts, you could do it like this:

```
int firstNumber, secondNumber;
...
double result = ((double)firstNumber)/secondNumber;
```

With the new casts, you'd write it this way:

```
double result = static_cast<double>(firstNumber)/secondNumber;
```

Now *there's* a cast that's easy to see, both for humans and for programs.

`static_cast` has basically the same power and meaning as the general-purpose C-style cast. It also has the same kind of restrictions. For example, you can't cast a `struct` into an `int` or a `double` into a pointer using `static_cast` any more than you can with a C-style cast. Furthermore, `static_cast` can't remove `const`ness from an expression, because another new cast, `const_cast`, is designed specifically to do that.

The other new C++ casts are used for more restricted purposes. `const_cast` is used to cast away the `const`ness or `volatileness` of an expression. By using a `const_cast`, you emphasize (to both humans and compilers) that the only thing you want to change through the cast is the `const`ness or `volatileness` of something. This meaning is enforced by compilers. If you try to employ `const_cast` for anything other than modifying the `const`ness or `volatileness` of an expression, your cast will be rejected. Here are some examples:

```
class Widget { ... };
class SpecialWidget: public Widget { ... };

void update(SpecialWidget *psw);

SpecialWidget sw;           // sw is a non-const object,
const SpecialWidget& csw = sw; // but csw is a reference to
                               // it as a const object

update(&csw);                // error! can't pass a const
                             // SpecialWidget* to a function
                             // taking a SpecialWidget*

update(const_cast<SpecialWidget*>(&csw));
                             // fine, the constness of &csw is
                             // explicitly cast away (and
                             // csw – and sw – may now be
                             // changed inside update)

update((SpecialWidget*)&csw);
                             // same as above, but using a
                             // harder-to-recognize C-style cast

Widget *pw = new SpecialWidget;

update(pw);                  // error! pw's type is Widget*, but
                             // update takes a SpecialWidget*

update(const_cast<SpecialWidget*>(pw));
                             // error! const_cast can be used only
                             // to affect constness or volatileness,
                             // never to cast down the inheritance
                             // hierarchy
```

By far the most common use of `const_cast` is to cast away the `const`ness of an object.

The second specialized type of cast, `dynamic_cast`, is used to perform *safe casts* down or across an inheritance hierarchy. That is, you use `dynamic_cast` to cast pointers or references to base class objects into pointers or references to derived or sibling base class objects in such a way that you can determine whether the casts succeeded.<sup>†</sup> Failed casts are indicated by a null pointer (when casting pointers) or an exception (when casting references):

```
Widget *pw;

...

update(dynamic_cast<SpecialWidget*>(pw));
           // fine, passes to update a pointer
           // to the SpecialWidget pw points to
           // if pw really points to one,
           // otherwise passes the null pointer

void updateViaRef(SpecialWidget& rsw);

updateViaRef(dynamic_cast<SpecialWidget&>(*pw));
           // fine, passes to updateViaRef the
           // SpecialWidget pw points to if pw
           // really points to one, otherwise
           // throws an exception
```

`dynamic_casts` are restricted to helping you navigate inheritance hierarchies. They cannot be applied to types lacking virtual functions (see also [Item 24](#)), nor can they cast away constness:

```
int firstNumber, secondNumber;

...

double result = dynamic_cast<double>(firstNumber)/secondNumber;
           // error! int has no virtual functions

const SpecialWidget sw;

...

update(dynamic_cast<SpecialWidget*>(&sw));
           // error! dynamic_cast can't cast
           // away constness
```

If you want to perform a cast on a type where inheritance is not involved, you probably want a `static_cast`. To cast constness away, you always want a `const_cast`.

The last of the four new casting forms is `reinterpret_cast`. This operator is used to perform type conversions whose result is nearly always implementation-defined. As a result, `reinterpret_casts` are rarely portable.

---

<sup>†</sup> A second, unrelated use of `dynamic_cast` is to find the beginning of the memory occupied by an object. We explore that capability in [Item 27](#).

The most common use of `reinterpret_cast` is to cast between function pointer types. For example, suppose you have an array of pointers to functions of a particular type:

```
typedef void (*FuncPtr)();           // a FuncPtr is a pointer
                                     // to a function taking no
                                     // args and returning void

FuncPtr funcPtrArray[10];           // funcPtrArray is an array
                                     // of 10 FuncPtrs
```

Let us suppose you wish (for some unfathomable reason) to place a pointer to the following function into `funcPtrArray`:

```
int doSomething();
```

You can't do what you want without a cast, because `doSomething` has the wrong type for `funcPtrArray`. The functions in `funcPtrArray` return `void`, but `doSomething` returns an `int`:

```
funcPtrArray[0] = &doSomething;    // error! type mismatch
```

A `reinterpret_cast` lets you force compilers to see things your way:

```
funcPtrArray[0] =                    // this compiles
    reinterpret_cast<FuncPtr>(&doSomething);
```

Casting function pointers is not portable (C++ offers no guarantee that all function pointers are represented the same way), and in some cases such casts yield incorrect results (see [Item 31](#)), so you should avoid casting function pointers unless your back's to the wall and a knife's at your throat. A sharp knife. A *very* sharp knife.

If your compilers lack support for the new casting forms, you can use traditional casts in place of `static_cast`, `const_cast`, and `reinterpret_cast`. Furthermore, you can use macros to approximate the new syntax:

```
#define static_cast(TYPE,EXPR)      ((TYPE) (EXPR))
#define const_cast(TYPE,EXPR)      ((TYPE) (EXPR))
#define reinterpret_cast(TYPE,EXPR) ((TYPE) (EXPR))
```

You'd use the approximations like this:

```
double result = static_cast(double, firstNumber)/secondNumber;
update(const_cast(SpecialWidget*, &sw));

funcPtrArray[0] = reinterpret_cast(FuncPtr, &doSomething);
```

These approximations won't be as safe as the real things, of course, but they will simplify the process of upgrading your code when your compilers support the new casts.



There is no easy way to emulate the behavior of a `dynamic_cast`, but many libraries provide functions to perform safe inheritance-based casts for you. If you lack such functions and you *must* perform this type of cast, you can fall back on C-style casts for those, too, but then you forego the ability to tell if the casts fail. Needless to say, you can define a macro to look like `dynamic_cast`, just as you can for the other casts:

```
#define dynamic_cast(TYPE,EXPR)      ((TYPE) (EXPR))
```

Remember that this approximation is not performing a true `dynamic_cast`; there is no way to tell if the cast fails.

I know, I know, the new casts are ugly and hard to type. If you find them too unpleasant to look at, take solace in the knowledge that C-style casts continue to be valid. However, what the new casts lack in beauty they make up for in precision of meaning and easy recognizability. Programs that use the new casts are easier to parse (both for humans and for tools), and they allow compilers to diagnose casting errors that would otherwise go undetected. These are powerful arguments for abandoning C-style casts, and there may also be a third: perhaps making casts ugly and hard to type is a *good* thing.

### Item 3: Never treat arrays polymorphically.

One of the most important features of inheritance is that you can manipulate derived class objects through pointers and references to base class objects. Such pointers and references are said to behave *polymorphically* — as if they had multiple types. C++ also allows you to manipulate *arrays* of derived class objects through base class pointers and references. This is no feature at all, because it almost never works the way you want it to.

For example, suppose you have a class `BST` (for binary search tree objects) and a second class, `BalancedBST`, that inherits from `BST`:

```
class BST { ... };

class BalancedBST: public BST { ... };
```

In a real program such classes would be templates, but that's unimportant here, and adding all the template syntax just makes things harder to read. For this discussion, we'll assume `BST` and `BalancedBST` objects contain only `ints`.

Consider a function to print out the contents of each `BST` in an array of `BSTs`:

```

void printBSTArray(ostream& s,
                  const BST array[],
                  int numElements)
{
    for (int i = 0; i < numElements; ++i) {
        s << array[i];           // this assumes an
    }                           // operator<< is defined
    }                           // for BST objects
}

```

This will work fine when you pass it an array of BST objects:

```

BST BSTArray[10];
...
printBSTArray(cout, BSTArray, 10);           // works fine

```

Consider, however, what happens when you pass `printBSTArray` an array of `BalancedBST` objects:

```

BalancedBST bBSTArray[10];
...
printBSTArray(cout, bBSTArray, 10);          // works fine?

```

Your compilers will accept this function call without complaint, but look again at the loop for which they must generate code:

```

for (int i = 0; i < numElements; ++i) {
    s << array[i];
}

```

Now, `array[i]` is really just shorthand for an expression involving pointer arithmetic: it stands for `*(array+i)`. We know that `array` is a pointer to the beginning of the array, but how far away from the memory location pointed to by `array` is the memory location pointed to by `array+i`? The distance between them is `i*sizeof(an object in the array)`, because there are `i` objects between `array[0]` and `array[i]`. In order for compilers to emit code that walks through the array correctly, they must be able to determine the size of the objects in the array. This is easy for them to do. The parameter `array` is declared to be of type `array-of-BST`, so each element of the array must be a `BST`, and the distance between `array` and `array+i` must be `i*sizeof(BST)`.

At least that's how your compilers look at it. But if you've passed an array of `BalancedBST` objects to `printBSTArray`, your compilers are probably wrong. In that case, they'd assume each object in the array is the size of a `BST`, but each object would actually be the size of a `BalancedBST`. Derived classes usually have more data members than their base classes, so derived class objects are usually larger than base class objects. We thus expect a `BalancedBST` object to be larger than a