C++ Gotchas

Avoiding Common Problems in Coding and Design

Stephen C. Dewhurst

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

C++ Gotchas

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.

C++ Gotchas

Avoiding Common Problems in Coding and Design

Stephen C. Dewhurst

Addison-Wesley

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

Library of Congress Cataloging-in-Publication Data
Dewhurst, Stephen C.
C++ gotchas : avoiding common problems in coding and design / Stephen C. Dewhurst.
p. cm—(Addison-Wesley professional computing series)
Includes bibliographical references and index.
ISBN 0-321-12518-5 (alk. paper)
1. C++ (Computer program language) I. Title. II. Series.

QA76.73.C153 D488 2002
005.13'3—dc21

2002028191

ISBN 0-321-12518-5

Text printed in the United States at Offset Paperback Manufacturers in Laflin, Pennsylvania.

9th Printing October 2008

To John Carolan

This page intentionally left blank

Contents

Preface

This book is the result of nearly two decades of minor frustrations, serious bugs, late nights, and weekends spent involuntarily at the keyboard. This collection consists of 99 of some of the more common, severe, or interesting C++ gotchas, most of which I have (I'm sorry to say) experienced personally.

The term "gotcha" has a cloudy history and a variety of definitions. For purposes of this book, we'll define C++ gotchas as common and preventable problems in C++ programming and design. The gotchas described here run the gamut from minor syntactic annoyances to basic design flaws to full-blown sociopathic behavior.

Almost ten years ago, I started including notes about individual gotchas in my C++ course material. My feeling was that pointing out these common misconceptions and misapplications in apposition to correct use would inoculate the student against them and help prevent new generations of C++ programmers from repeating the gotchas of the past. By and large, the approach worked, and I was induced to collect sets of related gotchas for presentation at conferences. These presentations proved to be popular (misery loves company?), and I was encouraged to write a "gotcha" book.

Any discussion of avoiding or recovering from C++ gotchas involves other subjects, most commonly design patterns, idioms, and technical details of C++ language features.

This is not a book about design patterns, but we often find ourselves referring to patterns as a means of avoiding or recovering from a particular gotcha. Conventionally, the pattern name is capitalized, as in "Template Method" pattern or "Bridge" pattern. When we mention a pattern, we describe its mechanics briefly if they're simple but delegate detailed discussion of patterns to works devoted to them. Unless otherwise noted, a fuller description of a pattern, as well as a richer discussion of patterns in general, may be found in Erich Gamma et al.'s *Design Patterns*. Descriptions of the Acyclic Visitor, Monostate, and Null Object patterns may be found in Robert Martin's *Agile Software Development*.

From the perspective of gotchas, design patterns have two important properties. First, they describe proven, successful design techniques that can be customized in a context-dependent way to new design situations. Second, and perhaps more

important, mentioning the application of a particular pattern serves to document not only the technique applied but also the reasons for its application and the effect of having applied it.

For example, when we see that the Bridge pattern has been applied to a design, we know at a mechanical level that an abstract data type implementation has been separated into an interface class and an implementation class. Additionally, we know this was done to separate strongly the interface from the implementation, so changes to the implementation won't affect users of the interface. We also know this separation entails a runtime cost, how the source code for the abstract data type should be arranged, and many other details.

A pattern name is an efficient, unambiguous handle to a wealth of information and experience about a technique. Careful, accurate use of patterns and pattern terminology in design and documentation clarifies code and helps prevent gotchas from occurring.

C++ is a complex programming language, and the more complex a language, the more important is the use of idiom in programming. For a programming language, an idiom is a commonly used and generally understood combination of lower-level language features that produces a higher-level construct, in much the same way patterns do at higher levels of design. Therefore, in C++ we can discuss copy operations, function objects, smart pointers, and throwing an exception without having to specify these concepts at their lowest level of implementation.

It's important to emphasize that an idiom is not only a common combination of language features but also a common set of expectations about how these combined features should behave. What do copy operations mean? What can we expect to happen when an exception is thrown? Much of the advice found in this book involves being aware of and employing idioms in C++ coding and design. Many of the gotchas listed here could be described simply as departing from a particular C++ idiom, and the accompanying solution to the problem could often be described simply as following the appropriate idiom (see Gotcha #10).

A significant portion of this book is spent describing the nuances of certain areas of the C++ language that are commonly misunderstood and frequently lead to gotchas. While some of this material may have an esoteric feel to it, unfamiliarity with these areas is a source of problems and a barrier to expert use of C++. These "dark corners" also make an interesting and profitable study in themselves. They are in C++ for a reason, and expert C++ programmers often find use for them in advanced programming and design.

Another area of connection between gotchas and design patterns is the similar importance of describing relatively simple instances. Simple patterns are important. In some respects, they may be more important than technically difficult patterns, because they're likely to be more commonly employed. The benefits obtained from the pattern description will, therefore, be leveraged over a larger body of code and design.

In much the same way, the gotchas described in this book cover a wide range of difficulty, from a simple exhortation to act like a responsible professional (Gotcha #12) to warnings to avoid misunderstanding the dominance rule under virtual inheritance (Gotcha #79). But, as in the analogous case with patterns, acting responsibly is probably more commonly applicable on a day-to-day basis than is the dominance rule.

Two common themes run through the presentation. The first is the overriding importance of convention. This is especially important in a complex language like C++. Adherence to established convention allows us to communicate efficiently and accurately with others. The second theme is the recognition that others will maintain the code we write. The maintenance may be direct, so that our code must be readily and generally understood by competent maintainers, or it may be indirect, in which case we must ensure that our code remains correct even as its behavior is modified by remote changes.

The gotchas in this book are presented as a collection of short essays, each of which describes a gotcha or set of related gotchas, along with suggestions for avoiding or correcting them. I'm not sure any book about gotchas can be entirely cohesive, due to the anarchistic nature of the subject. However, the gotchas are grouped into chapters according to their general nature or area of (mis)applicability.

Additionally, discussion of one gotcha inevitably touches on others. Where it makes sense to do so—and it generally does—I've made these links explicit. Cohesion within each item is sometimes at risk as well. Often it's necessary, before getting to the description of a gotcha, to describe the context in which it appears. That description, in turn, may require discussion of a technique, idiom, pattern, or language nuance that may lead us even further afield before we return to the advertised gotcha. I've tried to keep this meandering to a minimum, but it would have been dishonest, I think, to attempt to avoid it entirely. Effective programming in C++ involves intelligent coordination of so many disparate areas that it's impractical to imagine one can examine its etiology effectively without involving a similar eclectic collection of topics.

It's certainly not necessary—and possibly inadvisable—to read this book straight through, from Gotcha #1 to Gotcha #99. Such a concentrated dose of mayhem

may put you off programming in C++ altogether. A better approach may be to start with a gotcha you've experienced or that sounds interesting and follow links to related gotchas. Alternatively, you may sample the gotchas at random.

The text employs a number of devices intended to clarify the presentation. First, incorrect or inadvisable code is indicated by a gray background, whereas correct and proper code is presented with no background. Second, code that appears in the text has been edited for brevity and clarity. As a result, the examples as presented often won't compile without additional, supporting code. The source code for nontrivial examples is available from the author's Web site: www.semantics.org. All such code is indicated in the text by an abbreviated pathname near the code example, as in ➤➤ gotcha00/somecode.cpp.

Finally, a warning: the one thing you should not do with gotchas is elevate them to the same status as idioms or patterns. One of the signs that you're using patterns and idioms properly is that the pattern or idiom appropriate to the design or coding context will arise "spontaneously" from your subconscious just when you need it.

Recognition of a gotcha is analogous to a conditioned response to danger: once burned, twice shy. However, as with matches and firearms, it's not necessary to suffer a burn or a gunshot wound to the head personally to learn how to recognize and avoid a dangerous situation; generally, all that's necessary is advance warning. Consider this collection a means to keep your head in the face of C++ gotchas.

Stephen C. Dewhurst
Carver, Massachusetts
July 2002

Acknowledgments

Editors often get short shrift in a book's acknowledgments, sometimes receiving only a token ". . . and I also thank my editor, who surely must have been doing something while I was slaving over the manuscript." Debbie Lafferty, my editor, is responsible for the existence of this book. When I came to her with a mediocre proposal for a mediocre introductory programming text, she instead suggested expanding a section on gotchas into a book. I refused. She persisted. She won. Fortunately, Debbie is gracious in victory, and she has yet to utter an editorial "We told you so." Additionally, she surely must have been doing something while I slaved over the manuscript.

I would also like to thank the reviewers who lent their time and expertise to help make this a better book. Reviewing an unpolished manuscript is a time-consuming, often tedious, sometimes irritating, and nearly thankless task of professional courtesy (see Gotcha #12), and the reviewers' insightful and incisive comments were much appreciated. Steve Clamage, Thomas Gschwind, Brian Kernighan, Patrick McKillen, Jeffrey Oldham, Dan Saks, Matthew Wilson, and Leor Zolman contributed advice on technical issues and social propriety, corrections, code snippets, and an occasional snide remark.

Leor started review long before the manuscript was written, by sending me barbed comments on Web postings that were early versions of some of the gotchas appearing in this book. Sarah Hewins, my best friend and severest critic, earned both titles while reviewing various versions of the manuscript. David R. Dewhurst frequently put the entire project into perspective. Greg Comeau lent use of his marvelously standard C++ compiler for checking the code.

Like any nontrivial work about C++, this book is an amalgam of the work of many people. Over the years, many of my students, clients, and colleagues have augmented my unhappy facility for stumbling across C++ gotchas, and many of them have helped find solutions for them. While most of these contributions can no longer be acknowledged explicitly, it is possible to acknowledge more direct contributions:

The `Select` template of Gotcha #11 and the `OpNewCreator` policy of Gotcha #70 appear in Andrei Alexandrescu's *Modern C++ Design*.

I first encountered the problem of returning a reference to constant argument, described in Gotcha #44, in Cline et al.'s *C++ FAQs* (it began to appear in my clients' code immediately thereafter). Cline et al. also describe the technique mentioned in Gotcha #73 for circumventing overloaded virtual functions.

The `Cptr` template of Gotcha #83 is a modified version of the `CountedPtr` template that appeared in Nicolai Josuttis's *The C++ Standard Library*.

Scott Meyers has more to say about the improper overloading of operators `&&`, `||`, and `,`, described in Gotcha #14, in his *More Effective C++*. He describes in more detail the necessity of value return from a binary operator, as discussed in Gotcha #58, in *Effective C++* and describes the improper use of `auto_ptr`, treated in Gotcha #68, in *Effective STL*. The technique, mentioned in Gotcha #87, of returning a const from postfix increment and decrement operators is described in his *More Effective C++*.

Dan Saks presented the first cogent arguments I had heard for the forward declaration file approach described in Gotcha #8; he was also the first to identify the "Sergeant operator" of Gotcha #17, and he convinced me not to range-check increment and decrement on enum types, mentioned in Gotcha #87.

Herb Sutter's *More Exceptional C++*, Item 36, caused me to reread section 8.5 of the standard and update my understanding of formal argument initialization (see Gotcha #57).

Some of the material of Gotchas #10, #27, #32, #33, #38–#41, #70, #72–#74, #89, #90, #98, and #99 appeared in my "Common Knowledge" column that ran initially in *C++ Report* and later in *The C/C++ Users Journal*.

1 Basics

That a problem is basic does not mean it isn't severe or common. In fact, the common presence of the basic problems discussed in this chapter is perhaps more cause for alarm than the more technically advanced problems we discuss in later chapters. The basic nature of the problems discussed here implies that they may be present, to some extent, in almost all C++ code.

Gotcha #1: Excessive Commenting

Many comments are unnecessary. They generally make source code hard to read and maintain, and frequently lead maintainers astray. Consider the following simple statement:

```
a = b;  // assign b to a
```

The comment cannot communicate the meaning of the statement more clearly than the code itself, and so is useless. Actually, it's worse than useless. It's deadly. First, the comment distracts the reader from the code, increasing the volume of text the reader has to wade through in order to extract its meaning. Second, there is more source text to maintain, since comments must be maintained as the program text they describe is modified. Third, this necessary maintenance is often not performed.

```
c = b; // assign b to a
```

A careful maintainer cannot simply assume the comment is in error and is obliged to trace through the program to determine whether the comment is erroneous, officious (`c` is a reference to `a`), or subtle (assigning to `c` will later cause the same assignment to be propagated to `a` somehow). The line should originally have been written without a comment:

```
a = b;
```

The code is maximally clear as it stands, with no comment to be incorrectly maintained. This is similar in spirit to the well-worn observation that the most

efficient code is code that doesn't exist. The same applies to comments: the best comment is one that didn't have to be written, because the code it would otherwise have described is self-documenting.

Other common examples of unnecessary comments frequently occur in class definitions, either as the result of an ill-conceived coding standard or as the work of a C++ novice:

```
class C {
 // Public Interface
 public:
   C(); // default constructor
   ~C(); // destructor
   // . . .
};
```

You get the feeling you're reading someone's crib notes. If a maintainer has to be reminded of the meaning of the `public:` label, you don't want that person maintaining your code. None of these comments does anything for an experienced C++ programmer except clutter the code and provide more source text to be improperly maintained.

```
class C {
 // Public Interface
 protected:
   C( int ); // default constructor
 public:
   virtual ~C(); // destructor
   // . . .
};
```

Programmers also have a strong incentive not to "waste" lines of source text. Anecdotally, if a construct (function, public interface of a class, and so on) can be presented in a conventional and rational format on a single "page" of about 30–40 lines, it will be easy to understand. If it goes on to a second page, it will be about twice as hard to understand. If it goes onto a third page, it will be approximately four times as hard to understand.

A particularly odious practice is that of inserting change logs as comments at the head or tail of source code files:

```
/* 6/17/02 SCD fixed the gaforniflat bug */
```

Is this useful information, or is the maintainer just bragging? This comment is unlikely to be of any use whatever within a week or two of its insertion, but it will hang on grimly for years, distracting generations of maintainers. A much better alternative is to cede these commenting tasks to your version control software; a C++ source code file is no place to leave a laundry list.

One of the best ways to avoid comments and make code clear and maintainable is to follow a simple, well-defined naming convention and choose clear names that reflect the abstract meaning of the entity (function, class, variable, and so on) you're naming. Formal argument names in declarations are particularly important. Consider a function that takes three arguments of identical type:

```
/*
 Perform the action from the source to the destination.
 Arg1 is action code, arg2 is source, and arg3 is destination.
*/
void perform( int, int, int );
```

Not too terrible, but think what it would look like with seven or eight arguments instead of three. We can do better:

```
void perform( int actionCode, int source, int destination );
```

Better, though we should probably still have a one-liner that tells us what the function does (though not how it does it). One of the most attractive things about formal argument names in declarations is that they, unlike comments, are generally maintained along with the rest of the code, even though they have no effect on the code's meaning. I can't think of a single programmer who would switch the meanings of the second and third arguments of the `perform` function without also changing their names, but I can identify legions of programmers who would make the change without maintaining the comment.

Kathy Stark may have said it best in *Programming in C++*: "If meaningful and mnemonic names are used in a program, there is often only occasional need for additional comments. If meaningful names are not used, it is unlikely that any added comments will make the code easy to understand."

Another way to minimize comments is to employ standard or well-known components:

```
printf( "Hello, World!" ); // print "Hello, World" to the screen
```

This comment is both useless and only occasionally correct. It's not that standard components are necessarily self-documenting; it's that they're already well documented and well known.

```
swap( a, a+1 );
sort( a, a+max );
copy( a, a+max, ostream_iterator<T>(cout,"\n") );
```

Because `swap`, `sort`, and `copy` are standard components, additional comments inserted above can only clutter the source and introduce imprecision in the description of the standard operations.

Comments are not inherently harmful—and are often necessary—but they must be maintained, and they're typically harder to maintain than the code they document. Comments should not state the obvious or provide information better maintained elsewhere. The goal is not to eliminate comments at any cost but to employ the minimal volume of comments that permits the code to be readily understood and maintained.

Gotcha #2: Magic Numbers

Magic numbers, in the sense used here, are raw numeric literals used in contexts where named constants should be used instead:

```
class Portfolio {
   // . . .
   Contract *contracts_[10];
   char id_[10];
};
```

The main problem with magic numbers is that they have no semantic content to speak of; they are what they are. A `10` is a `10`, not a maximum number of contracts or the length of an identifier. Therefore we're obliged, when reading or maintaining code that employs magic numbers, to determine the intended meaning of each raw literal. That's work, and it's unnecessary and often inaccurate work.

For example, our poorly designed portfolio above can manage a maximum of ten contracts. That's not a lot of contracts, so we may decide to increase it to 32. (If we had any concern about safety and correctness, we'd use a standard `vector`.) The trouble is that we're now obliged to examine every source file that uses `Portfolio` for each occurrence of the literal `10`, to decide if that `10` means maximum number of contracts.

Actually, the situation can be worse. In large and long-lived projects, sometimes word gets out that the maximum number of contracts is ten, and this knowledge becomes embedded in code that doesn't even indirectly include the `Portfolio` header file:

```
for( int i = 0; i < 10; ++i )
   // . . .
```

Does this literal `10` refer to the maximum number of contracts? The length of an identifier? Something unrelated?

The chance confluence of raw literals can sometimes bring out the worst coding tendencies in programmers:

```
if( Portfolio *p = getPortfolio() )
   for( int i = 0; i < 10; ++i )
       p->contracts_[i] = 0, p->id_[i] = '\0';
```

Now the maintainer has to somehow tease apart the initializations of the different components of a `Portfolio` that would not have been combined but for the chance coincidence of the values of two distinct concepts. There is really no excuse for provoking all this complexity when the solution is so simple:

```
class Portfolio {
   // . . .
   enum { maxContracts = 10, idlen = 10 };
   Contract *contracts_[maxContracts];
   char id_[idlen];
};
```

Enumerators consume no space and cost nothing in runtime while providing clear, properly scoped names of the concepts for which they stand.

Less obvious disadvantages of magic numbers include the potential for imprecision in their types and the lack of associated storage. The type of the literal `40000`, for instance, is platform dependent. If the value 40000 can fit into an integer, its type is `int`. Otherwise, it's a `long`. If we don't want to leave ourselves open to obscure problems (like overload resolution ambiguities) when porting from platform to platform, it's probably best to say precisely what we mean rather than letting the compiler/platform combination decide for us:

```
const long patienceLimit = 40000;
```

Another potential problem with literals is that they have no address. This is not a common problem, but it is nevertheless occasionally useful to be able to point to or bind a reference to a constant:

```
const long *p1 = &40000; // error!
const long *p2 = &patienceLimit; // OK.
const long &r1 = 40000; // legal, but see Gotcha #44
const long &r2 = patienceLimit; // OK.
```

Magic numbers offer no advantage and many disadvantages. Use enumerators or initialized constants instead.

Gotcha #3: Global Variables

There is rarely an excuse for declaring a "raw" global variable. Global variables impede code reuse and make code hard to maintain. They impede reuse because any code that refers to a global variable is coupled to it and may not be reused without being accompanied by the global variable. They make code hard to maintain because it's difficult to determine what code is using a particular global variable, since any code at all has access to it.

Global variables increase coupling among components, because they often end up as a kind of primitive message-passing mechanism. Even if global variables work, it's often a practical impossibility to remove them from a large piece of software. If they work. Because global variables are essentially unprotected, any novice maintainer can trash the behavior of your global-dependent software at any time.

Users of global variables often cite convenience as a reason for using them. This is a fallacious or self-serving argument, because maintenance typically consumes more time than initial development, and use of global variables impedes maintenance. Suppose we have a system that requires access to a globally accessible "environment," of which (we're promised by our requirements) there is always exactly one. Unfortunately, we choose to use a global variable:

```
extern Environment * const theEnv;
```

Requirements live but to lie. Shortly before delivery, we'll find that the number of possible, simultaneous environments has increased to two. Or maybe three. Or maybe the number is set on startup. Or is totally dynamic. The usual last-minute change. In a large project with meticulous source-control procedures in place, it can be a time-consuming process to change every file, even in a minimal and

straightforward manner. It could take days or weeks. If we had avoided the use of a global variable, it would take five minutes:

```
Environment *theEnv();
```

Simply wrapping access in a function permits extension through the use of overloading or default argument initialization without the necessity of significant change to source code:

```
Environment *theEnv( EnvCode whichEnv = OFFICIAL );
```

Another, less obvious, problem with global variables is that they often require runtime static initialization. If a static variable's initial value can't be calculated at compile time, the initialization will take place at runtime, often with disastrous consequences (see Gotcha #55):

```
extern Environment * const theEnv = new OfficialEnv;
```

If a function or class guards access to the global information, the setting of the initial value can be delayed until it's safe to do so:

➤➤ gotcha03/environment.h

```
class Environment {
 public:
   static Environment &instance();
   virtual void op1() = 0;
   // . . .
 protected:
   Environment();
   virtual ~Environment();
 private:
   static Environment *instance_;
   // . . .
};
```

➤➤ gotcha03/environment.cpp

```
// . . .
Environment *Environment::instance_ = 0;

Environment &Environment::instance() {
   if( !instance_ )
       instance_ = new OfficialEnv;
   return *instance_;
}
```

In this case, we've employed a simple implementation of the Singleton pattern to perform lazy "initialization" (actually, to be technically precise, it's assignment) of the static environment pointer and thereby ensure that there is never more than a single `Environment` object. Note that `Environment` has no public constructor, so users of `Environment` must go through the `instance` member to gain access to the static pointer, allowing us to delay creation of the `Environment` object until the first request for access:

```
Environment::instance().op1();
```

More important, this controlled access provides flexibility to adapt the Singleton to future requirements without affecting existing source code. Later, if we go to a multithreaded design or decide to permit multiple environments, or whatever, we can modify the implementation of the Singleton, just as we modified the wrapper function earlier.

Avoid global variables. Safer and more flexible mechanisms are available to achieve the same results.

Gotcha #4: Failure to Distinguish Overloading from Default Initialization

Function overloading has little to do with default argument initialization. However, these two distinct language features are sometimes confused, because they can be used to produce interfaces whose syntax of use is similar. Nevertheless, the meanings of the interfaces are quite different:

➤➤ gotcha04/c12.h

```
class C1 {
 public:
   void f1( int arg = 0 );
   // . . .
};
```

➤➤ gotcha04/c12.cpp

```
// . . .
C1 a;
a.f1(0);
a.f1();
```

The designer of class `C1` has decided to employ a default argument initializer in the declaration of the operation `f1`. Therefore the user of `C1` has the option of

invoking the member function `f1` with an explicit single argument or with an implicit single argument of `0`. In the two calls to `C1::f1` above, the calling sequences produced are identical.

➤➤ gotcha04/c12.h

```
class C2 {
 public:
   void f2();
   void f2( int );
   // . . .
};
```

➤➤ gotcha04/c12.cpp

```
// . . .
C2 a;
a.f2(0);
a.f2();
```

The implementation of `C2` is quite different. The user has the choice of invoking two entirely different functions named `f2`, depending on the number of arguments passed. In our earlier example, the meanings of the two calls were identical. Here they're completely different, because they invoke different functions.

An even greater difference between the two interfaces is evident if we try to take the address of the class members `C1::f1` and `C2::f2`:

➤➤ gotcha04/c12.cpp

```
void (C1::*pmf)() = &C1::f1; //error!
void (C2::*pmf)() = &C2::f2;
```

With our implementation of class `C2`, the pointer to member `pmf` will refer to the `f2` that takes no argument. The variable `pmf` is a pointer to member function that takes no argument, so the compiler will correctly choose the first member `f2` as the initializer. With class `C1`, we'll get a compile-time error, because only one member function is named `f1`, and that function takes an integer argument.

Overloading is generally used to indicate that a set of functions has common abstract meaning but different implementations. Default initialization is generally used for convenience, to provide a simplified interface to a function. Overloading and default argument initializers are distinct language features with different intended purposes and behavior. Distinguish them carefully. (See also Gotchas #73 and #74.)

Gotcha #5: Misunderstanding References

There are two common problems with references. First, they're often confused with pointers. Second, they're underused. Many current uses of pointers in C++ are really C holdovers that should be ceded to references.

A reference is not a pointer. A reference is an alias for its initializer. Essentially, the only thing one can do with a reference is initialize it. After that, it's simply another way of referring to its initializer. (But see Gotcha #44.) A reference doesn't have an address, and it's even possible that it might not occupy any storage:

```
int a = 12;
int &ra = a;
int *ip = &ra; // ip refers to a
a = 42; // ra == 42
```

For this reason, it's illegal to attempt to declare a reference to a reference, a pointer to a reference, or an array of references. (Though the C++ standards committee has discussed allowing references to references in the future, at least in some contexts.)

```
int &&rri = ra; // error!
int &*pri; // error!
int &ar[3]; // error!
```

References can't be `const` or `volatile`, because aliases can't be `const` or `volatile`, though a reference can refer to an entity that is const or volatile. An attempt to declare a reference `const` or `volatile` directly is an error:

```
int &const cri = a; // should be an error . . .
const int &rci = a; // OK
```

Strangely, it's not illegal to apply a `const` or `volatile` qualifier to a type name that is of reference type. Rather than cause an error, in this case the qualifier is ignored:

```
typedef int *PI;
typedef int &RI;
const PI p = 0; // const pointer
const RI r = a; // just a reference!
```

There are no null references, and there are no references to `void`:

```
C *p = 0; // a null pointer
C &rC = *p; // undefined behavior
extern void &rv; // error!
```

A reference is an alias, and an alias has to refer to something.

Note, however, that a reference does not have to refer to a simple variable name. It's sometimes convenient to bind a reference to an lvalue (see Gotcha #6) resulting from a more complex expression:

```
int &el = array[n-6][m-2];
el = el*n-3;
string &name = p->info[n].name;
if( name == "Joe" )
   process( name );
```

A reference return from a function allows assignment to the result of a call. The canonical example of this is an index function for an abstract array:

➤➤ gotcha05/array.h

```
template <typename T, int n>
class Array {
 public:
   T &operator [](int i)
       { return a_[i]; }
   const T &operator [](int i) const
       { return a_[i]; }
   // . . .
 private:
   T a_[n];
};
```

The reference return permits a natural syntax for assignment to an array element:

```
Array<int,12> ia;
ia[3] = ia[0];
```

References may also be used to provide additional return values for functions:

```
Name *lookup( const string &id, Failure &reason );
// . . .
string ident;
// . . .
Failure reasonForFailure;
if( Name *n = lookup( ident, reasonForFailure ) ) {
   // lookup succeeded . . .
}
else {
   // lookup failed. check reason . . .
}
```

Casting an object to a reference type has a very different effect from the same cast to the nonreference version of the type:

```
char *cp = reinterpret_cast<char *>(a);
reinterpret_cast<char *&>(a) = cp;
```

In the first case, we're converting an integer into a pointer. (We're using `reinterpret_cast` in preference to an old-style cast, like `(char *)a`. See Gotcha #40.) The result is a copy of the integer's value, interpreted as a pointer.

The second cast is very different. The result of the cast to reference type is a reinterpretation of the integer object itself as a pointer. It's an lvalue, and we can assign to it. (Whether we will then dump core is another story. Use of `reinterpret_cast` generally implies "not portable.") An analogous attempt with a cast to nonreference will fail, because the result of the cast is an rvalue, not an lvalue:

```
reinterpret_cast<char *>(a) = 0; // error!
```

A reference to an array preserves the array bound. A pointer to an array does not:

```
int ary[12];
int *pary = ary; // point to first element
int (&rary)[12] = ary; // refer to whole array
int ary2[3][4];
int (*pary2)[4] = ary2; // point to first element
int (&rary2)[3][4] = ary2; // refer to whole array
```

This property can be of occasional use when passing arrays to functions. (See Gotcha #34.)

It's also possible to bind a reference to a function:

```
int f( double );
int (* const pf)(double) = f; // const pointer to function
int (&rf)(double) = f; // reference to function
```

There's not much practical difference between a constant pointer to function and a reference to function, except that the pointer can be explicitly dereferenced. As an alias, the reference cannot, although it can be converted implicitly into a pointer to function and then dereferenced:

```
a = pf( 12.3 ); // use pointer
a = (*pf)(12.3); // use pointer
a = rf( 12.3 ); // use reference
a = f( 12.3 ); // use function
a = (*rf)(12.3); // convert ref to pointer and deref
a = (*f)(12.3); // convert func to pointer and deref
```

Distinguish references and pointers.

Gotcha #6: Misunderstanding Const

The concept of constness in C++ is simple, but it doesn't necessarily correspond to our preconceived notions of a constant.

First, note the difference between a variable declared const and a literal:

```
int i = 12;
const int ci = 12;
```

The integer literal `12` is not a const. It's a literal. It has no address, and its value never changes. The integer `i` is an object. It has an address, and its value is variable. The const integer `ci` is also an object. It has an address, though (in this case) its value may not vary.

We say that `i` and `ci` may be used as lvalues, whereas the literal `12` may only be an rvalue. This terminology comes from the pseudoexpression `L = R`, indicating that an lvalue may appear as the left argument of an assignment and an rvalue

may appear only as the right argument of an assignment. However, this definition is not perfectly applicable in the case of C++ or standard C, where `ci` is an lvalue but may not be assigned to because it's a nonmodifiable lvalue. Consider lvalues as locations that may hold values, and rvalues as simple values with no associated address:

```
int *ip1 = &12; // error!
12 = 13; // error!
const int *ip2 = &ci; // OK
ci = 13; // error!
```

It's best to consider `const`, in the declaration of `ip2` above, a restriction on how we may manipulate `ci` through `ip2` rather than on how `ci` may be manipulated in general. Consider declaring a pointer to a const:

```
const int *ip3 = &i;
i = 10; // OK
*ip3 = 10; // error!
```

Here, we have a pointer to a constant integer that refers to a non-constant integer. The use of `const` in this case is simply a restriction on how `ip3` may be used. It doesn't imply that `i` won't change, only that we may not change it through `ip3`. Even subtler are combinations of `const` and `volatile`:

```
extern const volatile time_t clock;
```

The presence of the `const` qualifier indicates that we're not allowed to modify the variable `clock`, but the presence of the `volatile` qualifier indicates that the value of `clock` may (that is, will) change nonetheless.

Gotcha #7: Ignorance of Base Language Subtleties

Most C++ programmers are confident that they're fully familiar with what might be considered the C++ "base language": that part of C++ inherited from C. However, even experienced C++ programmers are sometimes ignorant of the more abstruse details of these basic C/C++ statements and operators.

The logical operators are not what one would ordinarily consider abstruse, but they seem to be increasingly underutilized by new C++ programmers. Isn't it irritating to see code like this?

```
bool r = false;
if( a < b )
   r = true;
```

Instead of this?

```
bool r = a<b;
```

Do you have to count to eight when presented with the following?

➤➤ gotcha07/bool.cpp

```
int ctr = 0;
for( int i = 0; i < 8; ++i )
   if( options & 1<<(8+i) )
       if( ctr++ ) {
           cerr << "Too many options selected";
           break;
       }
```

Instead of this?

➤➤ gotcha07/bool.cpp

```
typedef unsigned short Bits;
inline Bits repeated( Bits b, Bits m )
   { return b & m & (b & m)-1; }
// . . .
if( repeated( options, 0XFF00 ) )
   cerr << "Too many options selected";
```

What ever happened to Boolean logic?

Likewise, many programmers are ignorant of the fact that the result of a conditional operator is an lvalue (see Gotcha #6) if both its potential results are lvalues. This ignorance necessitates code like the following:

```
// version #1
if( a < b )
   a = val();
else if( b < c )
   b = val();
else
   c = val();
```

```
// version #2
a<b ? (a = val()) : b<c ? (b = val()) : (c = val());
```

An alternative solution with an lvalue conditional is definitely shorter and undeniably cooler:

```
// version #3
(a<b?a:b<c?b:c) = val();
```

While this piece of esoteric knowledge may not seem as immediately relevant as a sound appreciation of Boolean logic, many contexts in C++ allow only expressions (constructor member-initialization-lists, throw-expressions, and so on).

Additionally, note that the call to the entity `val` occurs multiple times in versions #1 and #2, whereas it appears only once in version #3. If `val` is a function, this is of little importance. However, if `val` is a preprocessor macro, the presence of multiple expansions may produce incorrect side effects (see Gotcha #26). In these contexts, the availability of an effective conditional operator as a substitute for an if-statement can be essential. Effectively, while I do not recommend that this construct be commonly used, I do recommend that it be commonly known. It should be available to the expert C++ programmer for those rare occasions when its use is required or preferable to other constructs. It's part of the C++ language for a reason.

Surprisingly, even the predefined index operator is often misunderstood. We all know that both array names and pointers may be indexed:

```
int ary[12];
int *p = &ary[5];
p[2] = 7;
```

The predefined index operator is just a shorthand for some pointer arithmetic and a dereference. The expression `p[2]` above is entirely equivalent to `*(p+2)`. Most C++ programmers with a C background are also aware that it's legal to use negative indexes, so the expression `p[-2]` is well defined and equivalent to `*(p-2)` or, if you prefer, `*(p+-2)`. However, it doesn't seem to be common knowledge that addition is commutative, since most C++ programmers are surprised to find that it's legal to index an integer with a pointer:

```
(-2)[p] = 6;
```

It's a simple transformation: `p[-2]` is equivalent to `*(p+-2)`, which is equivalent to `*(-2+p)`, which is equivalent to `(-2)[p]` (we need the parentheses because `[]` has higher precedence than unary minus).

What's the use of this bit of trivia? Well, for one thing, note that this commutativity of the index operator applies only to its predefined use with pointers. That is, if we see an expression like `6[p]`, we know we're dealing with the predefined index operator rather than with an overloaded member `operator []` (though `p` is not necessarily a pointer or array). It's also terrific when conversation lags at cocktail parties. However, before employing this syntax in production code, review Gotcha #11.

Most C++ programmers know that a switch-statement is pretty basic. They just don't know how basic. The abstract syntax of the switch-statement is simple:

```
switch( expression ) statement
```

The implications of this simple syntax are sometimes surprising.

Typically, the substatement that follows the switch expression is a block. Within the block is a set of case labels that implement basically a computed goto to a statement within the block. The first subtlety that new C and C++ programmers face is the concept of "fallthrough." That is, unlike many other modern programming languages, after a switch branches to the proper case label, its work is done. Where execution leads after that is totally up to the programmer:

```
switch( e ) {
default:
theDefault:
   cout << "default" << endl;
   // fallthrough . . .
case 'a':
case 0:
   cout << "group 1" << endl;
   break;
case max-15:
case Select<(MAX>12),A,B>::Result::value:
   cout << "group 2" << endl;
   goto theDefault;
}
```

Conventionally, whenever fallthrough is used on purpose—as opposed to its more typical inadvertent use—we insert a comment to indicate to future maintainers that we actually intended the fallthrough. Otherwise, maintainers have a tendency to insert inappropriate `break`s.

Note that the case labels must be integer constant-expressions. In other words, the compiler must be able to determine their values at compile time. However, as the somewhat flaky example above shows, there is quite a lot of leeway in how constant expressions may be defined. The case expression itself must be integral, or it may be an object with a conversion to an integral type. For example, `e` could be the name of a class object that declares a conversion operator to an integral type.

Note that the abstract syntax of the switch implies that it's even less structured than our example above implies. In particular, the case labels may appear anywhere within the switch-statement, and not necessarily at the same level:

```
switch( expr )
   default:
   if( cond1 ) {
       case 1: stmt1;
       case 2: stmt2;
   }
   else {
       if( cond2 )
           case 3:stmt2;
       else
           case 0: ;
   }
```

This may look a bit silly (it is, actually), but these more esoteric aspects of the base language can be useful on occasion. The above property of the switch, for instance, has been used to implement efficient external iteration of a complex data structure for a C++ compiler:

➤➤ gotcha07/iter.cpp

```
bool Postorder::next() {
  switch( pc )
  case START:
  while( true )
     if( !lchild() ) {
        pc = LEAF;
        return true;
```