原版风暴 • 深入 C++ 系列

PEARSON
Addison Wesley

堪与《Effective C++》、《More Effective C++》媲美
又一经典 C++ 设计与编程指南

C++ Gotchas（影印版）

Avoiding Common Problems in Coding and Design

[美] Stephen C. Dewhurst 著

Pearson Education

中国电力出版社
www.infopower.com.cn

原版风暴·深入C++系列

C++ Gotchas（影印版）

Avoiding Common Problems in Coding and Design

［美］Stephen C. Dewhurst 著

中国电力出版社

图书在版编目（CIP）数据

C++ Gotchas /（美）杜赫斯特著．影印本．—北京：中国电力出版社，2003
（原版风暴・深入 C++系列）
ISBN 7-5083-1494-8
Ⅰ.C...　Ⅱ.杜...　Ⅲ.C 语言－程序设计－英文　Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2003）第 023116 号

责任编辑：姚贵胜
丛 书 名：原版风暴・深入C++系列
书　　名：C++ Gotchas（影印版）
编　　著：（美）Stephen C. Dewhurst
出 版 者：中国电力出版社
地址：北京市三里河路6号　邮政编码：100044
电话：（010）88515918　　传真：（010）88423191
印　　刷：北京地矿印刷厂
开　　本：787×1092 1/16　　　**印　　张：**21.5
书　　号：ISBN 7-5083-1494-8
版　　次：2003年6月北京第一版
印　　次：2003年6月第一次印刷
定　　价：45.00 元

To John Carolan

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*.

C++ Gotchas

Contents

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 */
```