

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