What Every Professional C++ Programmer Needs to Know—Pared to Its Essentials So It Can Be Efficiently and Accurately Absorbed

C++ COMMON KNOWLEDGE

ESSENTIAL INTERMEDIATE PROGRAMMING

STEPHEN C. DEWHURST

C++ Common Knowledge

(continued inside the back cover)

Praise for C++ *Common Knowledge*

"We live in a time when, perhaps surprisingly, the best printed works on C++ are just now emerging. This is one of those works. Although C++ has been at the forefront of innovation and productivity in software development for more than two decades, it is only now being fully understood and utilized. This book is one of those rare contributions that can bear repeated study by practitioners and experts alike. It is not a treatise on the arcane or academic—rather it completes your understanding of things you think you know but will bite you sooner or later until you *really* learn them. Few people have mastered C++ and software design as well as Steve has; almost no one has such a level head as he when it comes to software development. He knows what you need to know, believe me. When he speaks, I always listen—closely. I invite you to do the same. You (and your customers) will be glad you did."

—Chuck Allison, editor, *The C++ Source*

"Steve taught me C++. This was back in 1982 or 1983, I think—he had just returned from an internship sitting with Bjarne Stroustrup [inventor of C++] at Bell Labs. Steve is one of the unsung heroes of the early days, and anything Steve writes is on my A-list of things to read. This book is an easy read and collects a great deal of Steve's extensive knowledge and experience. It is highly recommended."

—Stan Lippman, coauthor of C++ *Primer, Fourth Edition*

"I welcome the self-consciously non-Dummies approach of a short, smart book."

—Matthew P. Johnson, Columbia University

"I agree with [the author's] assessment of the types of programmers. I have encountered the same types in my experience as a developer and a book like this will go far to help bridge their knowledge gap.... I think this book complements other books, like *Effective* C++ by Scott Meyers. It presents everything in a concise and easy-to-read style."

—Moataz Kamel, senior software designer, Motorola Canada

"Dewhurst has written yet another very good book. This book should be required reading for people who are using C++ (and think that they already know everything in C++)."

—Clovis Tondo, coauthor of C++ *Primer Answer Book*

This page intentionally left blank

C++ Common Knowledge

This page intentionally left blank

C++ Common Knowledge

Essential Intermediate Programming

Stephen C. Dewhurst

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • 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++ common knowledge : essential intermediate programming / Stephen C. Dewhurst.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-32192-8 (pbk. : alk. paper)
1. C++ (Computer program language) I. Title.

QA76.73.C153D48797 2005
005.13'3—dc22

2004029089

ISBN 0-321-32192-8

Text printed in the United States at Demand Print Center in Old Tappan, New Jersey.

5th Printing July 2009

Contents

Preface

A successful book is not made of what is in it, but what is left out of it.

—Mark Twain

. . . as simple as possible, but no simpler.

—Albert Einstein

. . . a writer who questions the capacity of the person at the other end of the line is not a writer at all, merely a schemer.

—E.B. White

When he took over the editorship of the late *C++ Report*, the quick Herb Sutter asked me to write a column on a topic of my choosing. I agreed, and I chose to call the column "Common Knowledge." It was supposed to be, in Herb's words, "a regular summary of basic lore that every working C++ programmer should know—but can't always." After a couple of columns in that vein, however, I became interested in template metaprogramming techniques, and the topics treated in "Common Knowledge" from that point on were far from common.

However, the problem in the C++ programming industry that motivated my original choice of column remains. I commonly encounter the following types of individuals in my training and consulting work:

- Domain experts who are expert C programmers but who have only basic knowledge of (and perhaps some animosity toward) C++
- Talented new hires direct from university who have an academic appreciation for the C++ language but little production C++ experience

- Expert Java programmers who have little C++ experience and who have a tendency to program in C++ the way one would program in Java
- C++ programmers with several years of experience maintaining existing C++ applications but who have not been challenged to learn anything beyond the basics required for maintenance

I want to be immediately productive, but many of the people with whom I'm working or who I'm training require preliminary education in various C++ language features, patterns, and coding techniques before we can get down to business. Worse, I suspect that most C++ code is written in ignorance of at least some of these basics and is therefore not what most C++ experts would consider to be production quality.

This book addresses this pervasive problem by providing essential, common knowledge that every professional C++ programmer needs to know, in a form that is pared to its essentials and that can be efficiently and accurately absorbed. Much of the information is already available from other sources or is part of that compendium of unwritten information that all expert C++ programmers know. The advantage is that this material resides in one place and was selected according to what my training and consulting experience over many years has shown are the most commonly misunderstood and most useful language features, concepts, and techniques.

Perhaps the most important aspect of the sixty-three short items that make up this book is what they leave out, rather than what they contain. Many of these topics have the potential to become complex. An author's ignorance of these complexities could result in an uninformed description that could mislead the reader, but an expert discussion of a topic in its full complexity could inundate the reader. The approach used here is to filter out *needless* complexity in the discussion of each topic. What remains, I hope, is a clear distillation of the essentials required for production C++ programming. C++ language wonks will recognize, therefore, that I've left out discussion of some issues that are interesting and even important from a theoretical perspective, but the ignorance of which does not commonly affect one's ability to read and write production C++ code.

Another motivation for this book came as I was engaged in conversation with a group of well-known C++ experts at a conference. There was a general pall or depression among these experts that modern C++ is so complex that the "average" programmer can no longer understand it. (The specific issue was name binding in the context of templates and namespaces. Yes, getting worked up about such a topic does imply the need for more play with normal children.) On reflection, I'd have to say our attitude was pretentious and our gloom unwarranted. We "experts" have no such problems, and it's as easy to program in C++ as it is to speak a (vastly more complex) natural language, even if you can't diagram the deep structure of your every utterance. A recurring theme of this book is that while the full description of the minutia of a particular language feature may be daunting, day-to-day use of the feature is straightforward and natural.

Consider function overloading. A full description occupies a large chunk of the standard and whole or multiple chapters in many C++ texts. And yet, when faced with

```
void f( int );
void f( const char * );
//…
f( "Hello" );
```

not a single practicing C++ programmer will be unable to determine which `f` is called. Full knowledge of the rules by which a call to an overloaded function is resolved is useful but only rarely necessary. The same applies to many other ostensibly complex areas of C++ language and idiom.

This is not to say that all the material presented here is easy; it's "as simple as possible, but no simpler." In C++ programming, as in any other worthwhile intellectual activity, many important details can't be written on an index card. Moreover, this is not a book for "dummies." I feel a great deal of responsibility to those who grant a portion of their valuable time to reading my books. I respect these readers and try to communicate with them as I would in person to any of my colleagues. Writing at an eighth-grade level to a professional isn't writing. It's pandering.

Many of the book's items treat simple misunderstandings that I've seen over and over again, which just need to be pointed out (for example, scope order for member function lookup and the difference between overriding and overloading). Others deal with topics that are in the process of becoming essential knowledge for C++ professionals but are often incorrectly assumed to be difficult and are avoided (for example, class template partial specialization and template template parameters). I've received some criticism from the expert reviewers of the manuscript that I've spent too much space (approximately one third of the book) on template issues that are not really common knowledge. However, each of these experts pointed out one, two, or several of the template topics they thought did belong in the book. The telling observation is, I think, that there was little overlap among these suggestions, and every template-related item had at least one supporter.

This is the crux of the issue with the items that make up this book. I don't expect any reader to be ignorant of every item's topic, and it's likely that some readers will be familiar with all of them. Obviously, if a reader is not familiar with a particular topic, there would be (I presume) some benefit in reading about it. However, even if a reader is already familiar with a topic, I'd hope that reading about it from a new perspective might clear up a slight misunderstanding or lead to a deeper understanding. This book may also have a role in saving the more experienced C++ programmer precious time. Competent C++ programmers often find themselves (as described previously) answering the same questions over and over again to the detriment of their own work. I'd suggest that the approach of "read this first, and *then* let's talk" would save these C++ gurus countless hours and direct their expertise instead to the complex problems for which it's really needed.

I initially tried to group these sixty-three items into neat chapters, but the items had other ideas. They instead tended to clump themselves together in ways that ranged from the obvious to the unexpected. For example, the items related to exceptions and resource management form a rather natural group. Less obviously, the items *Capability Queries*, *Meaning of Pointer* Comparison, *Virtual Constructors and Prototype*, *Factory Method*, and *Covariant Return Types* are strongly and somewhat surprisingly interrelated and are best grouped in close proximity to each other. *Pointer Arithmetic* decided to hang with *Smart Pointers* rather than with the pointer and array material earlier in the book. Rather than attempt to impose an arbitrary chapter structure on these natural groupings, I decided to grant the individual items freedom of association. Of course, many other interrelationships exist among the topics treated by the items than can be represented in a simple linear ordering, so the items make frequent internal references among themselves. It's a clumped but connected community.

While the main idea is to be brief, discussion of a topic sometimes includes ancillary details that are not directly related to the subject at hand. These details are never necessary to follow the discussion, but the reader is put on notice that a particular facility or technique exists. For instance, the `Heap` template example that appears in several items informs the reader in passing about the existence of the useful but rarely discussed STL heap algorithms, and the discussion of placement new outlines the technical basis of the sophisticated buffer management techniques employed by much of the standard library. I also try to take the opportunity, whenever it seems natural to do so, to fold the discussion of subsidiary topics into the discussion of a particular, named item. Therefore, *RAII* contains a short discussion of the order of constructor and destructor activation, *Template Argument Deduction* discusses the use of helper functions for specializing class templates, and *Assignment and Initialization Are Different* folds in a discussion of computational constructors. This book could easily have twice the number of items, but, like the clumping of the items themselves, correlation of a subsidiary topic with a specific item puts the topic in context and helps the reader to absorb the material efficiently and accurately.

I've reluctantly included several topics that cannot reasonably be treated in this book's format of short items. In particular, the items on design patterns and the design of the standard template library are laughably short and incomplete. Yet they make an appearance simply to put some common misconceptions to rest, emphasize the importance of the topics, and encourage the reader to learn more.

Stock examples are part of our programming culture, like the stories that families swap when they gather for holidays. Therefore, `Shape`, `String`, `Stack`, and many of the other usual suspects put in an appearance. The common appreciation of these baseline examples confers the same efficiencies as design patterns in communication, as in "Suppose I want to rotate a `Shape`, except…" or "When you concatenate two `Strings`…" Simply mentioning a common example orients the conversation and avoids the need for time-consuming background discussion. "You know how your brother acts whenever he's arrested? Well, the other day…"

Unlike my previous books, this one tries to avoid passing judgment on certain poor programming practices and misuses of C++ language features; that's a goal for other books, the best of which I list in the bibliography. (I was, however, not entirely successful in avoiding the tendency to preach; some bad programming practices just have to be mentioned, even if only in passing.) The goal of this book is to inform the reader of the technical essentials of production-level C++ programming in as efficient a manner as possible.

.

—Stephen C. Dewhurst
Carver, Massachusetts
January 2005

Acknowledgments

Peter Gordon, editor *on ne peut plus extraordinaire*, withstood my kvetching about the state of education in the C++ community for an admirably long time before suggesting that I do something about it. This book is the result. Kim Boedigheimer somehow managed to keep the entire project on track without even once making violent threats to the author.

The expert technical reviewers—Matthew Johnson, Moataz Kamel, Dan Saks, Clovis Tondo, and Matthew Wilson—pointed out several errors and many infelicities of language in the manuscript, helping to make this a better book. A stubborn individual, I haven't followed *all* their recommendations, so any errors or infelicities that remain are entirely my fault.

Some of the material in this book appeared, in slightly different form, in my "Common Knowledge" column for *C/C++ Users Journal*, and much of the material appeared in the "Once, Weakly" Web column on semantics.org. I received many insightful comments on both print and Web articles from Chuck Allison, Attila Fehér, Kevlin Henney, Thorsten Ottosen, Dan Saks, Terje Slettebø, Herb Sutter, and Leor Zolman. Several in-depth discussions with Dan Saks improved my understanding of the difference between template specialization and instantiation and helped me clarify the distinction between overloading and the appearance of overloading under ADL and infix operator lookup.

This book relies on less direct contributions as well. I'm indebted to Brandon Goldfedder for the algorithm analogy to patterns that appears in the item on design patterns and to Clovis Tondo both for motivation and for his assistance in finding qualified reviewers. I've had the good fortune over the years to teach courses based on Scott Meyers's *Effective C++*, *More Effective C++*, and *Effective STL* books. This has allowed me to observe firsthand what background information was commonly missing from students who wanted to profit from these industry-standard, intermediate-level C++ books, and those observations have helped to shape the set of topics treated in this book. Andrei Alexandrescu's work inspired me to experiment with template metaprogramming rather than do what I was supposed to be doing, and both Herb Sutter's and Jack Reeves's work with exceptions has helped me to understand better how exceptions should be employed.

I'd also like to thank my neighbors and good friends Dick and Judy Ward, who periodically ordered me away from my computer to work the local cranberry harvest. For one whose professional work deals primarily in simplified abstractions of reality, it's intellectually healthful to be shown that the complexity involved in convincing a cranberry vine to bear fruit is a match for anything a C++ programmer may attempt with template partial specialization.

Sarah G. Hewins and David R. Dewhurst provided, as always, both valuable assistance and important impediments to this project.

I like to think of myself as a quiet person of steady habits, given more to calm reflection than strident demand. However, like those who undergo a personality transformation once they're behind the wheel of an automobile, when I get behind a manuscript I become a different person altogether. Addison-Wesley's terrific team of behavior modification professionals saw me through these personality issues. Chanda Leary-Coutu worked with Peter Gordon and Kim Boedigheimer to translate my rantings into rational business proposals and shepherd them through the powers-that-be. Molly Sharp and Julie Nahil not only turned an awkward word document into the graceful pages you see before you, they managed to correct many flaws in the manuscript while allowing me to retain my archaic sentence structure, unusual diction, and idiosyncratic hyphenation. In spite of my constantly changing requests, Richard Evans managed to stick to the schedule and produce not one, but two separate indexes. Chuti Prasertsith designed a gorgeous, cranberry-themed cover. Many thanks to all.

A Note on Typographical Conventions

As mentioned in the preface, these items frequently reference one another. Rather than simply mention the item number, which would force an examination of the table of contents to determine just what was being referenced, the title of the item is italicized and rendered in full. To permit easy reference to the item, the item number and page on which it appears are appended as subscripts. For example, the item referenced *Eat Your Vegetables* [64, 256] tells us that the item entitled "Eat Your Vegetables" is item 64, which can be found on page 256.

Code examples appear in fixed-width font to better distinguish them from the running text. Incorrect or inadvisable code examples appear with a gray background, and correct and proper code appears with no background.

Item 1 Data Abstraction

A "type" is a set of operations, and an "abstract data type" is a set of operations with an implementation. When we identify objects in a problem domain, the first question we should ask about them is, "What can I do with this object?" not "How is this object implemented?" Therefore, if a natural description of a problem involves employees, contracts, and payroll records, then the programming language used to solve the problem should contain `Employee`, `Contract`, and `PayrollRecord` types. This allows an efficient, two-way translation between the problem domain and the solution domain, and software written this way has less "translation noise" and is simpler and more correct.

In a general-purpose programming language like C++, we don't have application-specific types like `Employee`. Instead, we have something better: the language facilities to create sophisticated abstract data types. The purpose of an abstract data type is, essentially, to extend the programming language into a particular problem domain.

No universally accepted procedure exists for designing abstract data types in C++. This aspect of programming still has its share of inspiration and artistry, but most successful approaches follow a set of similar steps.

1. Choose a descriptive name for the type. If you have trouble choosing a name for the type, you don't know enough about what you want to implement. Go think some more. An abstract data type should represent a single, well-defined concept, and the name for that concept should be obvious.
2. List the operations that the type can perform. An abstract data type is defined by what you can do with it. Remember initialization (constructors), cleanup (destructor), copying (copy operations), and conversions (nonexplicit single-argument constructors and conversion operators). Never, ever, simply provide a bunch of get/set operations on the data members of the implementation. That's not data abstraction; that's laziness and lack of imagination.
3. Design an interface for the type. The type should be, as Scott Meyers tells us, "easy to use correctly and hard to use incorrectly." An

abstract data type extends the language; do proper language design. Put yourself in the place of the user of your type, and write some code with your interface. Proper interface design is as much a question of psychology and empathy as technical prowess.

4. Implement the type. Don't let the implementation affect the interface of the type. Implement the contract promised by the type's interface. Remember that the implementations of most abstract data types will change much more frequently than their interfaces.

Item 2 | Polymorphism

The topic of polymorphism is given mystical status in some programming texts and is ignored in others, but it's a simple, useful concept that the C++ language supports. According to the standard, a "polymorphic type" is a class type that has a virtual function. From the design perspective, a "polymorphic object" is an object with more than one type, and a "polymorphic base class" is a base class that is designed for use by polymorphic objects.

Consider a type of financial option, `AmOption`, as shown in Figure 1.

An `AmOption` object has four types: It is simultaneously an `AmOption`, an `Option`, a `Deal`, and a `Priceable`. Because a type is a set of operations (see *Data Abstraction* [1, 1] and *Capability Queries* [27, 93]), an `AmOption` object can be manipulated through any one of its four interfaces. This means that an `AmOption` object can be manipulated by code that is written to the `Deal`, `Priceable`, and `Option` interfaces, thereby allowing the implementation of `AmOption` to leverage and reuse all that code. For a polymorphic type such as `AmOption`, the most important things inherited from its base classes are their interfaces, not their implementations. In

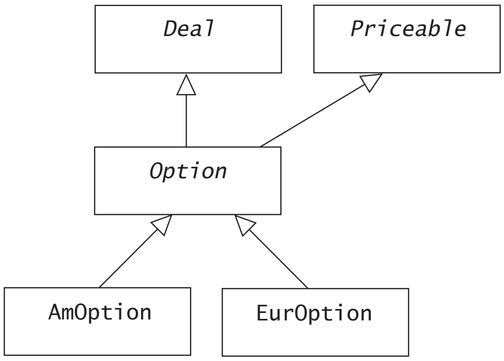

Figure 1 | Polymorphic leveraging in a financial option hierarchy. An American option has four types.

fact, it's not uncommon, and is often desirable, for a base class to consist of nothing but interface (see *Capability Queries* [27, 93]).

Of course, there's a catch. For this leveraging to work, a properly designed polymorphic class must be substitutable for each of its base classes. In other words, if generic code written to the `Option` interface gets an `AmOption` object, that object had better behave like an `Option`!

This is not to say that an `AmOption` should behave identically to an `Option`. (For one thing, it may be the case that many of the `Option` base class's operations are pure virtual functions with no implementation.) Rather, it's profitable to think of a polymorphic base class like `Option` as a contract. The base class makes certain promises to users of its interface; these include firm syntactic promises that certain member functions can be called with certain types of arguments and less easily verifiable semantic promises concerning what will actually occur when a particular member function is called. Concrete derived classes like `AmOption` and `EurOption` are subcontractors that implement the contract `Option` has established with its clients, as shown in Figure 2.

For example, if `Option` has a pure virtual `price` member function that gives the present value of the `Option`, both `AmOption` and `EurOption` must implement this function. It obviously won't implement identical behavior for these two types of `Option`, but it should calculate and return a price, not make a telephone call or print a file.

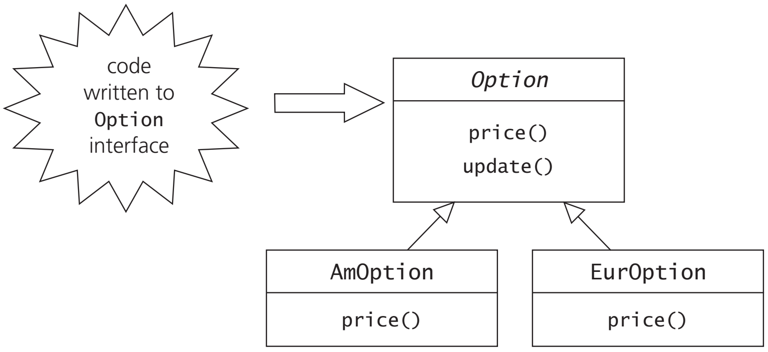

Figure 2 | A polymorphic contractor and its subcontractors. The `Option` base class specifies a contract.

On the other hand, if I were to call the `price` function of two different interfaces to the *same* object, I'd better get the same result. Essentially, either call should bind to the same function:

```
AmOption *d = new AmOption;
Option *b = d;
d->price(); // if this calls AmOption::price...
b->price(); // ...so should this!
```

This makes sense. (It's surprising how much of advanced object-oriented programming is basic common sense surrounded by impenetrable syntax.) If I were to ask you, "What's the present value of that American option?" I'd expect to receive the same answer if I'd phrased my question as, "What's the present value of that option?"

The same reasoning applies, of course, to an object's nonvirtual functions:

```
b->update(); // if this calls Option::update...
d->update(); // ...so should this!
```

The contract provided by the base class is what allows the "polymorphic" code written to the base class interface to work with specific options while promoting healthful ignorance of their existence. In other words, the polymorphic code may be manipulating `AmOption` and `EurOption` objects, but as far as it's concerned they're all just `Option`s. Various concrete `Option` types can be added and removed without affecting the generic code that is aware only of the `Option` base class. If an `AsianOption` shows up at some point, the polymorphic code that knows only about `Option`s will be able to manipulate it in blissful ignorance of its specific type, and if it should later disappear, it won't be missed.

By the same token, concrete option types such as `AmOption` and `EurOption` need to be aware only of the base classes whose contracts they implement and are independent of changes to the generic code. In principle, the base class can be ignorant of everything but itself. From a practical perspective, the design of its interface will take into account the requirements of its anticipated users, and it should be designed in such a way that derived classes can easily deduce and implement its contract (see *Template Method* [22, 77]). However, a base class should have no specific knowledge of any of the classes derived from it, because such knowledge inevitably makes it difficult to add or remove derived classes in the hierarchy.

In object-oriented design, as in life, ignorance is bliss (see also *Virtual Constructors and Prototype* [29, 99] and *Factory Method* [30, 103]).