Advanced C++

PROGRAMMING STYLES AND IDIOMS

JAMES O. COPLIEN

BIBLIOTHEQUE DU CERIST

Advanced C++
Programming Styles and Idioms

James O. Coplien
AT&T Bell Laboratories

ADDISON-WESLEY PUBLISHING COMPANY

Reading, Massachusetts • Menlo Park, California • New York • Don Mills, Ontario
Wokingham, England • Amsterdam • Bonn • Sydney • Singapore
Tokyo • Madrid • San Juan • Milan • Paris

BIBLIOTHEQUE DU CERIST

BIBLIOTHEQUE DU CERIST

Library of Congress Cataloging-in-Publication Data

Coplien, James O.
Advanced C++ programming styles and idioms / James O. Coplien.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-54855-0
1. C++ (Computer program language) I. Title.
QA76.73.C153C67 1992
005.26'2--dc20

91-19806
CIP

AT&T

Copyright © 1992 by AT&T Bell Telephone Laboratories, Incorporated.
Reprinted with corrections January, 1992
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

This book was typeset in Times, Courier, and Helvetica by the author, using a Linotronic 200P phototypsetter and an Amdahl 5990 running the UNIX® System V Operating System.

Ada is a Registered Trademark of the United States Department of Defense. Apollo is a Registered Trademark of Hewlett-Packard Company. Microsoft Windows is a Trademark of Microsoft, Inc. Simula 67 is a Trademark of Simula AS. Smalltalk-80 is a Trademark of ParcPlace Systems. Sun is a Trademark of Sun Microsystems. MacApp is a Trademark of Apple Computer, Inc. UNIX is a Registered Trademark of UNIX System Laboratories. X Windows, The X Window System are Trademarks of the Massachusetts Institute of Technology. XWin is a Registered Trademark of UNIX System Laboratories.

1 2 3 4 5 6 7 8 9 10-HA-9594939291

To Sandra,
Christopher,
Lorelei, and
Andrew Michael
with love

S. D. G.

BIBLIOTHEQUE DU CERIST

BIBLIOTHEQUE DU CERIST

Obtaining Copies of the Book's Software Examples

The software for many of the book's examples has been made available on-line, on system `research.att.com`, where copies may be retrieved for personal, non-commercial use. The source can be retrieved by establishing an `ftp` connection to the `research` machine, using the login `netlib` and using your electronic mail address as the password. The files appear under the directory `c++/idioms`. Alternatively, the software can be retrieved through electronic mail (if you do not have `ftp` access) by sending electronic mail messages of the form

```
send index for c++/idioms
send 2-2a.c 2-2b.c 2-4.c 2-5.c from c++/idioms
```

to the login `netlib@research.att.com`. The `index` file lists all available files.

BIBLIOTHEQUE DU CERIST

Preface

This book is designed to help programmers who have already learned C++ develop their programming expertise. To understand how programmers achieve proficiency, we need to understand not only how people learn a new language (such as a programming language), but also how a language is used to solve software problems effectively.

Learning Programming Languages

Not everything you need to know about a product is described in the owner's manual. Before the arrival of our first child, my wife and I were admonished by a friend that no book, and no training, could completely prepare us for the art of parenting. We must of course learn minimal, essential skills. But the interesting, challenging, and rewarding aspects of raising a child go beyond this basic knowledge. For example, no book or ''owner's manual'' will help you understand why your three-year-old daughter rubs toothpaste in your one-year-old's hair, or why your children hang their socks in the refrigerator.

The same is true of programming languages. Programming language syntax shapes our thinking to a degree, but what we learn in the ''owner's manual'' about syntax alone only gets us started. Most of what guides the structure of our programs, and therefore of the systems we build, is the *styles* and *idioms* we adopt to express design concepts.

Style distinguishes excellence from accomplishment. An effective parenting style, or programming style, comes from personal experience or by building on the experience of others. A software engineer who knows how to match a programming language to the needs of an application, writes excellent programs. To achieve this level of expertise, we need to go beyond rules and rote, into convention and style, and ultimately into abstractions of concept and structure. It is in that sense that this book is "advanced."

The rules, conventions, and concepts of programming drive the structure of the systems we build: They give us a model of how to build systems. A model for problem decomposition and system composition is a *paradigm*, a pattern for dividing the world into manageable parts. C++ is a multiparadigm language. C programmers use C++ as a better C. Object-oriented advocates do everything polymorphically. In fact, a variety of approaches is usually necessary to express the solution to a software problem efficiently and elegantly.

Learning a programming language is much like learning a natural language. Knowledge of basic syntax lets a programmer write simple procedures and build them into nontrivial programs, just as someone with a vocabulary of a few hundred German words can write a story far richer than see-Dick-run. But mastery of language is quite another issue. That such stories are nontrivial does not make them elegant or demonstrate fluency. Learning language syntax and basic semantics is like taking a 13-hour course in German: It prepares you for the task of ordering a bratwurst, but not for going to Germany to make a living, and certainly not for getting a job as a German language journalist or poet. The difference is in learning the *idioms* of the language. For example, there is nothing in C itself that establishes

```
while (*cp1++ = *cp2++);
```

as a fundamental building block, but a programmer unfamiliar with this construct would not be perceived as a fluent C programmer.

In programming, as in natural language, important idioms underly the suitability and expressiveness of linguistic constructs even in everyday situations. Good idioms make the application programmer's job easier, just as idioms in any language enrich communication. Programming idioms are reusable "expressions" of programming semantics, in the same sense that classes are reusable units of design and code. Simple idioms (like the `while` loop above) are notational conveniences, but seldom are central to program design. This book focuses on idioms that influence how the language is used in the overall structure and implementation of a design. Such idioms take insight and time to refine, more so than the simple notational idioms. The idioms usually involve some intricacy and complexity, details that can be written once and stashed away. Once established, programming language idioms can be used with convenience and power.

BIBLIOTHEQUE DU CERIST

The Book's Approach

Assuming a background in the basic syntax of C++, this book imparts the proficiency that expert C++ programmers gain through experience by giving a feel for the styles and idioms of the language. It shows how different styles let C++ be used for simple data abstraction, full-fledged abstract data type implementation, and various styles of object-oriented programming. It also explores idioms that the core of the C++ language does not directly support, such as functional and frame-based programming, and advanced garbage collection techniques.

BIBLIOTHEQUE DU CERIST

The Book's Structure

Rather than taking a ''flat'' approach to learning the advanced features of C++ by organizing around language features, this book looks at these increasingly powerful abstractions from the perspective of the C++ features required to support them. Each chapter of this book is organized around a family of such idioms. The idioms progressively build on each other in successive chapters.

Chapter 1 provides a historical perspective on C++ idioms. It provides some motivation as to why idioms came about, and varying degrees to which different idioms can be thought of as part of the language or as outside the language.

Chapter 2 introduces the fundamental C++ language building blocks: classes and member functions. Though much of the material is basic, the chapter establishes idioms and vocabulary that recur in later chapters. It introduces compiler type systems, and their relationship to user-defined types and classes, from a design perspective. It also presents idioms that make `const` more useful.

Chapter 3 introduces idioms that make classes ''complete'' types. C++ has been evolving to automate more and more of the work of copying and initializing objects, but programmers still need to customize assignment and default constructors for most nontrivial classes. This chapter provides a framework for that customization. I call the idioms described in this chapter *canonical forms*, meaning that they define principles and standards to make the underlying mechanics of objects work. In addition to the most commonly used canonical form, idioms are presented to apply reference counting to new and existing classes. These are the first idioms of the book to go beyond straightforward application of the base C++ syntax. A variation on reference counting, counted pointers, is shown as a way to move C++ a step further away from the machine, abandoning pointers in deference to smarter, pointer-like objects. Lastly, the chapter looks at how to separate the creation of an object from its initialization. To someone familiar with basic C++, this might seem an unnatural idiom: C++ tightly couples these two operations. The need to separate them arises in the design of device drivers and in systems with mutually dependent resources.

Chapter 4 introduces inheritance; Chapter 5 adds polymorphism to inheritance to introduce object-oriented programming. Many new C++ programmers get "inheritance fever," using it at every occasion. While it is true that inheritance is used mostly to support the object paradigm, it has a distinctly separate application for software reuse. Introducing inheritance apart from polymorphism helps the reader separate the two concepts and avoids the confusion that often arises from trying to internalize two foreign concepts at once.

Chapter 6 approaches the constructs, styles and idioms of C++ from the perspectives of architecture and design. It examines what classes *mean* at the level of an application, high above the level of syntax. Appreciating the relationships between the design abstractions of an application, and between the classes and objects of its implementation, leads to systems that are robust and easily evolved. Another key to evolution is broadening designs beyond a specific application, to cover applications for a whole domain; guiding principles for domain analysis are an important part of this chapter. The chapter contains numerous rules of thumb about appropriate use of inheritance, an area of difficulty for inexpert C++ programmers. Readers who have been exposed to object-oriented design will appreciate the explanation in this chapter of how to transform the output of design to C++ code. Encapsulation as an alternative to inheritance, both for reuse and for polymorphism, is explored in the context of the C++ language.

Chapter 7 explores reuse of code and designs. Four distinct code mechanisms are explored, with particular attention devoted to the benefits and pitfalls of "inheritance fever." Idioms are presented to significantly reduce the code volume generated by parameterized type libraries using templates.

The remainder of the book stretches beyond native C++ into advanced programming idioms. Chapter 8 introduces exemplars, objects that take over many of the roles of C++ classes. Exemplars are presented as special objects that solve some common development problems, such as the "virtual constructor" problem. But exemplars also lay the groundwork for more powerful design techniques supporting class independence and independent development.

Chapter 9, which focuses on symbolic language styles, breaks with concepts many hold fundamental to C++ programming including strong typing and explicit memory management. The idioms of this chapter are certainly outside mainstream C++ development and are reminiscent of styles found in Smalltalk and Lisp. One might claim that those who want to program in Smalltalk should program in Smalltalk. Those who want Smalltalk in all of its glory should indeed use Smalltalk. However, the fact that the styles presented in this chapter are exotic does not mean that the need

BIBLIOTHEQUE DU CERIST

that the need for them is rare or esoteric. Sometimes we want a small portion of a system to have the flexibility and polymorphism of symbolic languages, and in those situations we need to step outside the bounds of the C++ philosophy while working in the confines of the C++ semantics and type system. This chapter regularizes such idioms so they do not have to be created from scratch each time they are needed.

Chapter 9 also presents idioms supporting incremental run-time update. Implementations of this idiom are necessarily dependent on many details of the target platform. The gist of this material is to familiarize the reader with the level of technology at which incremental loading issues must be worked. The example presented is typical and, as such techniques go, it is neither obtuse nor trivial. The code presented for incremental loading needs major reworking for platforms other than Sun work stations, and it may be found altogether unsuitable to some environments. None of the book's other idioms depend on this idiom, so it can be pursued or rejected on its own merits. The goal of Chapter 9 is *not* to change C++ into Smalltalk; this cannot, and should not, be done. These idioms are less compile-time type safe and generally less efficient than "native C++" code; what they offer is flexibility and an increased measure of automated memory management.

Chapter 10 covers dynamic multiple inheritance. Multiple inheritance is a controversial C++ feature, and discussion of this dynamic variation is separated out to avoid tainting other chapters. While static multiple inheritance as described in Chapter 5 has value, dynamic multiple inheritance avoids problems of a combinatorial explosion of class combinations. This approach has been found valuable in many real-life programs including editors, CAD systems, and database managers.

The last chapter discusses objects from a high-level, system view. The chapter raises the level of abstraction above chunks the size of a C++ class, to larger and more encompassing units of software architecture, organization, and administration. The chapter puts a number of important system issues in perspective, including scheduling, exception handling, and distributed processing. Some guidelines for modularization and reuse are also discussed, tying together the concepts of Chapters 6 and 7. Included in this discussion are considerations for library structure and maintenance.

In Appendix A, the basic C++ concepts are compared with their C analogues. Many readers will have already learned these basics or can find them in introductory texts. This material is included here for two reasons. First, it serves as a ready reference for those occasions when you need clarification of an obscure construct without having to go to a separate text. Second, C and C++ styles are viewed from a design perspective, showing how to mix procedural and object-oriented code. This is particularly important for C++ programmers working with a base of C code.

The examples in this book are based on Release 3.0 of C++, and have been tried under Release 3 of the AT&T USL Language System on many different hardware

BIBLIOTHEQUE DU CERIST

platforms, and under some other C++ environments as well. Many of the examples have been tried under GNU C++ Version 1.39.0 and Zortech C++ 2.0, though examples using features of the 3.0 release await forthcoming releases of these tools. Some code makes use of general purpose class libraries for maps, sets, lists, and others. Efficient versions of such libraries are available from many vendors, and adequately functional versions can be created from scratch for pedagogical purposes. The skeletons, and sometimes complete bodies, of many general-purpose classes can be gleaned from examples in the book. Key class names are listed in the Index.

BIBLIOTHEQUE DU CERIST

Acknowledgments

This book owes much to many friends. The original impetus for the book came from Chris Carson at Bell Laboratories, and he if anyone is the book's godfather. I appreciate his initiative and support in starting this effort. The book reflects the guidance of Keith Wollman, my resourceful and accommodating editor, and Helen Wythe, the production supervisor. Lorraine Mingacci collaborated significantly with me on Chapter 6, and discussions with her about the rest of the book have been of immense value. The book owes much to a core team of thorough and relentless reviewers and contributors of ideas: Tim Born, Margaret A. Ellis, Bill Hopkins, Andrew Koenig, Stan Lippman, Barbara Moo, Bonnie Prokopowicz, Larry Schutte, and Bjarne Stroustrup. Alexis Layton, Jim Adcock, and Mike Ackroyd offered suggestions and insights to keep the book focused and balanced, and I am deeply grateful for their contributions. Many other improvements are owed to reviews by Miron Abramovici, Martin Carroll, Brian Kernighan, Andrew Klein, Doug McIlroy, Dennis Mancl, Warren Montgomery, Tom Mueller, Anil Pal, Peggy Quinn, and Ben Rovegno. More than anyone else, Brett L. Schuchert is responsible for the improvements in the second printing. Mary Crabb, Jean Owen, and Chris Scussel lent their expertise on the wonders of text formatting. Credit goes to Corporate Desktop Services of Glen Ellyn, Illinois, for working out the final phototypesetting bugs. Many thanks to Judy Marxhausen for advice on special topics.

Much credit goes to AT&T managers for their encouragement and for their provision of time and resources to work on the book. Thanks to Paul Zislis and Ben Rovegno for support in the early days, and very special thanks to Warren Montgomery, Jack Wang, and Eric Sumner, Jr., for their support, ideas, and forbearance.

Students from many of my courses have been a good source of input and feedback on materials that would later be brought together in this book. Special thanks go to the students of the C++ courses I taught at AT&T Bell Laboratories, both in Naperville, Illinois and in Columbus, Ohio, in 1989.

BIBLIOTHEQUE DU CERIST

Contents

BIBLIOTHEQUE DU CERIST

BIBLIOTHEQUE DU CERIST

BIBLIOTHEQUE DU CERIST

BIBLIOTHEQUE DU CERIST