Books

C++ Primer, 2nd edition, Stanley B. Lippman, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1991. 614 pp. (ISBN 0-201-54848-8).

In his introduction, Lippman recounts how, when faced with updating his book to keep up with advances in C++, he resolved to make it "... a better edition, not just an update." The general response from colleagues was a hesitant "Well, [just] don't ruin it"! To his credit, Lippman has succeeded in making the new edition even better than the first.

Programmers moving from C will likely appreciate that the book's general organization parallels (at least initially) the better C texts in that it starts building up from the basic elements of the language, rather than starting with philosophy. The complete C++ language is covered including templates, with an abbreviated section on exception handling.

The complex rules for resolving type conversions and overloaded functions are explained particularly well. Lippman sticks to the practice of clearly listing the sequence of steps that compilers (and hopefully, programmers) use in determining whether code is valid. Rather than having to read through long sections of prose, the reader will find the pertinent information conveniently summarized by these time-saving lists.

Readers coming from a C background tend to be more accepting of language features when they can "kick the tires" and figure out how the underlying mechanisms work. Lippman provides some insight into this in the discussion of derived classes and virtual base classes. Unfortunately, there is not enough detail for those who desire to know how multiple inheritance can be implemented efficiently or how virtual function tables actually work.

Lippman sidesteps a number of potential pitfalls and manages to stick to the most important features of the C++ language. The emphasis is on getting the reader up-to-speed on using C++. As such, completeness is sacrificed on occasion, but never enough to be of concern to the vast majority of readers.

Templates provide a mechanism for the programmer to express the workings of a class, while remaining somewhat independent of actual data types. The compiler takes care of generating a version of the class for each required type. Lippman demonstrates this using a `queue` class. He shows how a single description (template) for the `queue` class is used to create variations of the queue for integers, strings, or other data types. The important topic of auxiliary (or helper) classes is also discussed.

Two chapters devoted to object-oriented design successfully stress some of the important points in selecting and refining the key objects for an application. The example used to show an application of the techniques (adding a template capability to a C++ compiler) becomes complex enough at times so as to obscure the intended lessons.

An introduction to the C++ I/O streams library is included as well as an appendix contrasting C++ with ANSI C. The preprocessor goes largely unnoticed. Structures (i.e., `structs`) are not mentioned, presumably with the aim of encouraging use of the `class` keyword.

Lippman describes the current (3.0) level of C++ throughout this book. He also helpfully indicates where this behavior may differ from what has

been initially adopted by the ANSI C++ committee. An appendix itemizes changes from the 2.0 level of C++.

Overall, Lippman has again succeeded in producing an excellent starting point for programmers wanting to move into C++. This book is highly recommended for the first step into C++ and object-oriented programming.

The C++ Programming Language, 2nd edition, Bjarne Stroustrup, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1991. 669 pp. (ISBN 0-201-53992-6).

The first edition of this book was released in 1986 as the reference guide for the C++ language. Since then, a large body of experience in using C++ for a variety of projects has been acquired. The C++ language itself has also evolved significantly. Stroustrup has greatly enhanced this new edition to take both into account.

The C++ language is covered in its entirety, including thorough descriptions of both templates and exception handling. Stroustrup's treatment of the language tends to be more detailed than other C++ books. Sprinkled among this additional detail, are interesting tidbits about *why* a feature was designed to work as it does, in addition to the usual *how*. Often, the *why* proves at least as useful as the *how*.

The discussion of correctly using exceptions is very thorough. Exceptions provide an alternative to the "`errno`" error-handling strategy of times past. Stroustrup describes exceptions as an "alternate return mechanism," in that a function can either return a correct result, or can trigger (or *throw*) an exception. Possible exceptions may now be listed on a function prototype. Stroustrup even covers how to have your own function(s) called when either an exception is not being handled, or a thrown exception was not declared in a function's interface. A caution is that exceptions are a relatively new feature, so not all C++ compilers support them yet.

Organizationally, this book is well suited to quickly locating information related to a specific feature. Beware though, that the index has quirks such as listing "`virtual`" and "virtual" as different headings, even though they differ only in typeface. The examples tend to be of a manageable size and are presented in a style amenable to directly transplanting them into your own programs without having to bring along a lot of extra baggage.

The C++ reference manual is provided as an appendix. This is the document that the ANSI C++ committee is using as a starting point in their standardization efforts. It serves as an important adjunct to the text since information that is omitted in the interest of readability is covered by the appendix.

Like Lippman's book, this is not the place to come to learn the finer points of the C/C++ preprocessor. While the reference manual appendix provides an adequate description of the preprocessor, the text avoids use of the preprocessor (except for `#include` directives). This is probably based on the belief that with the changes to the `const` qualifier (borrowed from ANSI C), templates, and the ability to create functions that are expanded in-line, macros are largely unnecessary.

A chapter on "dos" and "don'ts" in designing C++ libraries gives some guidance on topics ranging from determining which entities should be objects, to the factors in choosing inheritance versus containment when designing classes. Also insightful are sections contrasting the problems of small-scale and large-scale development efforts. Attention is paid to differentiating between the usual problems of scale and those introduced by a switch to an object-oriented methodology and C++.

A section on run-time type information looks at how such a capability can be "added to" C++ without having to extend the language. Descriptions of both the uses and abuses of such a capability are discussed.

C++ I/O streams are described in detail. Information ranges from how to properly overload the "`>>`" and "`<<`" operators, to how the internal buffering of data is handled. Coverage is complete enough for the reader to grasp how the different stream classes interact, and what to consider when subclassing to create new I/O classes. I/O manipulators are also described.

Overall, Stroustrup succeeds in balancing the interests of C programmers that are looking to move up into C++, against those who actually look upon C++ as a completely new language that just happens to have some syntactic similarities to C. The book will prove useful for both. In terms of presentation style, if you liked Kernighan and Ritchie, you will be happy with this one.

Advanced C++, Programming Styles and Idioms, James O. Coplien, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992. 520 pp. (ISBN 0-201-54855-0).

As the title suggests, this book is intended for a different audience than the two reviewed above. The reader is expected to already have a working knowledge of C++. The author concentrates less on describing language features and more on schemes for exploiting them.

The material is presented essentially in three phases: (1) creating safe, efficient C++ data types (classes); (2) object-oriented (OO) design and reuse; and (3) programming using meta-objects.

The first sections cover what goes into creating a new class that is able to fit well with existing types, and still be used efficiently. Coplien introduces what he terms *canonical forms* as a guideline for the minimum interface that a class should provide to fit well into the C++ type system (e.g., a copy constructor, a default constructor, a virtual destructor, etc.). Preventing multiple constructor calls for the same object via an extra level of constructor indirection is detailed. The problem arises with multiple inheritance in conjunction with virtual base classes.

Lower-level efficiency issues are considered next, starting with how to replace a class's `new` and `delete` operators to speed object allocation/deallocation. Reference counting (managing multiple objects sharing a single copy of data) and *smart* pointers (overloading of the -> operator to yield a different type) are covered. How to use these ideas with existing classes (where you might not be at liberty to make changes to the class itself) is described.

Object-oriented design and reuse are tackled next with an emphasis on surrogate objects as a way to insulate class clients from internal changes. Coplien uses the analogy of letters and envelopes, with envelope classes that *stand-in* (or conceal) one or more letter classes. Changes to the letter (including a possible change of its type) need not affect those who only see the envelope. The impacts of deciding to structure classes in this fashion are thoroughly discussed.

Coplien coins the term *exemplars* for what are referred to as *factory objects* or *meta-objects* in other languages. The basis of the approach is that rather than dynamically managing C++ objects the usual way (via `new/delete`), the programmer addresses the requests to a controller object for that class. Coplien shows some of the advantages in adopting such a scheme, such as allowing for garbage collection.

One of the more interesting areas is class "versioning"—what happens when the implementation of a class changes? What is the impact on its clients? This problem has received very limited attention up until now in the C++ community. The answer has so far been that clients can simply recompile their programs. As software becomes larger and class libraries are increasingly interdependent, forced recompilations become less satisfactory. Coplien identifies a number of the pertinent issues and suggests a scheme to solve part of the problem. Code is given to show a way to dynamically load a new class definition into a running program, which also involves evolving active objects.

A downside to the many interesting ideas presented is that a number of them cannot be used in isolation. The garbage collection scheme presented, while ably highlighting design decisions involved, largely restricts how the objects are to be used. You really need to be fully committed to the use of exemplars in order to make use of a number of the suggestions. On the other side of the coin, these same illustrations serve to show the burden that such features can impose on performance and structure of classes.

An appendix is provided contrasting C and C++. Unfortunately, the C dialect indicated most often is pre-ANSI C, thereby painting a somewhat dated picture of the differences.

Overall, this is a worthwhile book for those that have been using C++ for a while and would like

to see how common idioms like reference counting can be implemented. In the larger view, while a number of techniques presented in the book are characterized as enabling a *symbolic* style of programming, they seem intended to enable the programmer to escape C++'s strong typing. Coplien delves into enough detail throughout to allow readers to understand the implications of such an escape and to evaluate for themselves what they must give up in one area to gain in another. If you are an intermediate-to-experienced C++ programmer, this book is definitely worth reading.

Shawn Elliott
IBM Canada Laboratory
Toronto
Ontario

Client/Server Programming with OS/2 2.0, second edition, Robert Orfali and Dan Harkey, Van Nostrand Reinhold, New York, 1992. 1,112 pp. (ISBN 0-442-01219-5).

The whole is more than the sum of its parts—something everyone knows—and yet, when it comes to technology and its use, we tend to only explain the parts. Take client/server applications as an example: within a single basic application, you use communications, a database, and a graphical user interface. Then there are application-level design decisions to be made, like which communications protocol is best, how much processing goes into the client versus the server, should you use static SQL or dynamic SQL, should you use remote SQL or stored procedures, and what can a transaction monitor do for you. This is a lot of ground to cover!

This explains why Orfali and Harkey's new book, *Client/Server Programming with OS/2 2.0,* second edition, is 1,112 pages long. This is the first comprehensive book on client/server computing using Operating System/2* (OS/2*) 2.0 as a base. It is a book that really brings all the pieces together. Orfali and Harkey first introduce client/server concepts in a tutorial-like style, then demonstrate through working code the design tradeoffs involved in integrating database servers, LANs, and client workstations that use OS/2's new object-oriented user interface (OOUI).

Orfali and Harkey have made this book very approachable, with a friendly style, many illustrations, and a useful index. This is a good thing in a book that weighs a full kilogram. If you read the best-selling first edition of the book, you will want a copy of the second. It has 600 new pages that tailor the book to OS/2 2.0.

In this bible of a book, there is a great deal for everyone. The first 200 pages is a "book within a book." The authors provide a useful overview of client/server architectures—including the Distributed Computing Environment, multiservers, open systems, and Systems Application Architecture*. This section also serves as an introduction to the new OS/2 2.0 features, how OS/2 2.0 compares to other client/server platforms (Windows** 3.X, UNIX**, NT**, and NetWare**). The authors review a dozen related, OS/2-based client/server products—including Multimedia Presentation Manager/2, Database Manager*, Distributed Database Connection Services/2*, Communications Manager*, TCP/IP for OS/2, LAN Server, and NetWare Requester for OS/2 2.0.

The back 920 pages are a system architect's paradise. They cover almost every tradeoff in the design of client/server systems. Wherever possible Orfali and Harkey put together some clever benchmarks (with working code) to make their point:

- In Part II the authors go over the design tradeoffs involved in creating multitasking clients and servers. Which are better, threads or processes? What are the the overheads associated with semaphores versus interprocess communication mechanisms such as Named Pipes?
- In Part III the authors cover the tradeoffs in peer-to-peer and client/server LAN communications. To make this hard topic fun, they stage a BLOB Olympics using a BLOB (binary large objects) server application that compares the performance of NetWare, LAN Server, CPI-C over APPC, NetBIOS, and sockets over TCP/IP.
- In Parts IV and V the authors develop a program that implements the TP1 transaction benchmark and use it to demonstrate how client/server database design tradeoffs impact application performance. Which would you design: fat clients or fat servers? You will know after this section.
- In Parts VI and VII the authors explore the benefits of object-oriented user interfaces (OOUIs) using System Object Model (SOM) and the Workplace Shell* (WPS) class library. They demonstrate the benefits of OOUIs over graph-

ical user interfaces (GUIs) with a flamboyant example of a Club Med** Reservation System. The Club Med front end is created using six new objects derived from SOM/WPS classes. The back end uses a transaction server built on top of the OS/2 Database Manager.

In conclusion, this book's real strength is that it transcends the armchair discussion of client/server and gets into real-life design tradeoffs using working examples. While this book will primarily appeal to the OS/2 2.0 client/server crowd, its strong OS/2 orientation (and advocacy) should not deter client/server theorists with other OS affiliations. There are important practical lessons in this book—such as fat clients versus fat servers—that apply equally well to OS/2, NT, UNIX, or Tandem Computers, Inc.'s Guardian** OS. The section on OOUIs and distributed objects will be of interest to all providers of visual front ends to databases.

Lee C. Chang
Professor, College of Engineering
San Jose State University
California

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Microsoft Corporation, UNIX Systems Laboratories, Inc., Novell Corporation, Club Med Sales, Inc., or Tandem Computers, Inc.

Note—The books received are those the Editor thinks might be of interest to our readers. The reviews express the opinions of the reviewers.