
Preface

Welcome to ANSI/ISO Standard C++! This book is by an old guy and a young guy. The old guy (HMD; Massachusetts Institute of Technology 1967) has been programming and/or teaching programming for 39 years. The young guy (PJD; MIT 1991) has been programming for 18 years and has caught the teaching and writing “bug.” The old guy programs and teaches from experience; the young guy does so from an inexhaustible reserve of energy. The old guy wants clarity; the young guy wants performance. The old guy appreciates elegance and beauty; the young guy wants results. We got together to produce a book we hope you will find informative, interesting and entertaining.

These are exciting times in the C++ community with the approval of the ANSI/ISO C++ Standard. ANSI (the American National Standards Institute) and ISO (the International Standards Organization), have cooperated to develop what has become one of the most important worldwide standards for the computing community.

When we wrote the second edition of *C++ How to Program*, we aimed the book at college-level courses which at the time were primarily being taught in Pascal or C, emphasizing the procedural programming paradigm. Writing a C++ textbook for the Computer Science I and II audiences presented a difficult challenge to us. We would need to describe two programming paradigms, both procedural programming (because C++ still includes C) and object-oriented programming. This practically doubled the amount of material that would need to be presented at the introductory level. We chose a strategy of presenting the C-like material on primitive data types, control structures, functions, arrays, pointers, strings and structures in the first five chapters of the book. We then presented object-oriented programming in Chapters 6 through 15.

C++ How to Program became the most widely used college-level C++ textbook in the world. We delayed writing the new edition for two reasons:

1. C++ was under active development over this time period, with new drafts of the standards document appearing on a regular basis, but with no clear signs from the standards committee that the draft standard was going to be accepted “as is” within a short period of time.

©1994–2000 by Deitel & Associates, Inc. and Prentice Hall. All rights reserved.

2. We were waiting for a key sign that it was time for a new edition of *C++ How to Program*. That came in July 1997 with the publication of Bjarne Stroustrup's third edition of his book *The C++ Programming Language: Third Edition*. Stroustrup created C++ and his books are the definitive works on the language. At this point, we felt that the "new definition" of C++ was sufficiently stable for us to publish *C++ How to Program: Second Edition*.

We diverted our attention for a time to produce five Java publications. But the excitement of the impending acceptance of the ANSI/ISO C++ Draft Standard drew our attention back to C++.

C++ How to Program: Third Edition

We performed an extensive review process on this *Third Edition* that led to thousands of polishing changes. We also completely updated the programs in the text to conform to the C++ standard's use of namespaces.

The major new feature of this *Third Edition* is a complete, fully-implemented case study on object-oriented design using the Unified Modeling Language™ (UML). We felt that a commitment to larger-scale object-oriented design projects is something that has been lacking in introductory programming textbooks. This optional case study is highly recommend because it will considerably enhance the students' experience in a first-year university programming sequence. This case study provides students with an opportunity to immerse themselves in a 1000+ line C++ program that was carefully scrutinized by a team of distinguished industry and academic reviewers.

In the previous editions of this book, we included special "Thinking About Objects" sections at the ends of Chapters 1 through 7. These sections walked the student through the steps needed to design the software simulator for an elevator system. We asked the student to complete these steps and to implement their design in C++. For *C++ How to Program: Third Edition*, we have completely remodeled this case study. At the ends of Chapters 1 through 7 and the end of Chapter 9, we use the "Thinking About Objects" sections to present a carefully paced introduction to object-oriented design using the UML. The UML is now the most widely used graphical representation scheme for modeling object-oriented systems. The UML is a complex, feature-rich graphical language. In our "Thinking About Objects" sections, we present a concise, simplified subset of these features. We then use this subset to guide the reader through a first design experience with the UML intended for the novice object-oriented designer/programmer. We present this case study in a fully solved format. This is not an exercise; rather, it is an end-to-end learning experience that concludes with a detailed walkthrough of the C++ code.

In each of the first five chapters we concentrate on the "conventional" methodology of structured programming, because the objects we will build will be composed, in part, of structured-program pieces. We then end each chapter with a "Thinking About Objects" section in which we present an introduction to object orientation using the Unified Modeling Language (UML). Our goal in these "Thinking About Objects" sections is to help students develop an object-oriented way of thinking, so they can immediately put to use the object-oriented programming concepts that they begin learning in Chapter 6. In the first of these sections at the end of Chapter 1, we introduce basic concepts (i.e., "object think") and terminology (i.e., "object speak"). In the optional "Thinking About Objects" sections at the

ends of Chapters 2 through 5 we consider more substantial issues as we attack a challenging problem with the techniques of object-oriented design (OOD). We analyze a typical problem statement that requires a system to be built, determine the objects needed to implement that system, determine the attributes the objects will need to have, determine the behaviors these objects will need to exhibit and specify how the objects will need to interact with one another to meet the system requirements. We do all this even before we discuss how to write object-oriented C++ programs. In the “Thinking About Objects” sections at the ends of Chapters 6, 7 and 9, we discuss a C++ implementation of the object-oriented system we designed in the earlier chapters.

This case study is significantly larger than any other project attempted in the book. We feel that the student gains significant experience by following this complete design and implementation process. This project forced us to incorporate topics that we do not discuss in any other section of the book, including object interaction, an in-depth discussion of handles, the philosophy of using references vs. pointers and the use of forward declarations to avoid the circular include problem. This case study will help prepare students for the kinds of substantial projects encountered in industry.

“Thinking About Objects” Sections

In Chapter 2, we begin the first phase of an object-oriented design (OOD) for the elevator simulator—identifying the classes needed to implement the simulator. We also introduce the UML use case, class and object diagrams and the concepts of associations, multiplicity, composition, roles and links.

In Chapter 3, we determine many of the class attributes needed to implement the elevator simulator. We also introduce the UML statechart and activity diagrams and the concepts of events and actions as they relate to these diagrams.

In Chapter 4, we determine many of the operations (behaviors) of the classes in the elevator simulation. We also introduce the UML sequence diagram and the concept of messages sent between objects.

In Chapter 5, we determine many of the collaborations (interactions between objects in the system) needed to implement the elevator system and represent these collaborations using the UML collaboration diagram. We also include a bibliography and a list of Internet and World Wide Web resources that contain the UML 1.3 specifications and other reference materials, general resources, tutorials, FAQs, articles, whitepapers and software.

In Chapter 6, we use the UML class diagram developed in previous sections to outline the C++ header files that define our classes. We also introduce the concept of handles to objects in the system and we begin to study how to implement handles in C++.

In Chapter 7, we present a complete elevator simulator C++ program (approximately 1000 lines of code) and a detailed code walkthrough. The code follows directly from the UML-based design created in previous sections and employs our good programming practices, including the use of **static** and **const** data members and functions. We also discuss dynamic-memory allocation, composition and object interaction via handles and how to use forward declarations to avoid the circular include problem.

In Chapter 9, we update the elevator simulation design and implementation to incorporate inheritance. We also suggest further modifications so that the student may then design and implement, using the tools presented in the previous sections.

We sincerely hope that this newly updated elevator simulation case study provides a challenging and meaningful experience for both students and instructors. We employ a carefully developed, incremental object-oriented process to produce a UML-based design for our elevator simulator. From this design, we produce a substantial working C++ implementation using key programming notions, including classes, objects, encapsulation, visibility, composition and inheritance. We would be most grateful if you would take a moment to send your comments, criticisms and suggestions for improving this case study to us at deitel@deitel.com.

C++ How to Program: Third Edition Ancillary Package

We have worked hard to produce a textbook and ancillaries that we hope you and your students will find valuable. The following ancillary resources are available:

- *C++ How to Program: Third Edition's 268 program examples* are included on the CD-ROM in the back of the textbook. This helps instructors prepare lectures faster and helps students master C++. The examples are also available for download at www.deitel.com. When extracting the source code from the ZIP file, you must use a ZIP-file reader such as WinZip (<http://www.winzip.com/>) or PKZIP (<http://www.pkware.com/>) that understands directories. The file should be extracted into a separate directory (e.g., `cpphttp3e_examples`).
- *Microsoft Visual C++ 6 Introductory Edition software* is provided on the textbook's CD-ROM. This software allows students to edit, compile and debug C++ programs. We have provided at no charge a short Visual C++ 6 tutorial (in Adobe PDF format) on our Web site (www.deitel.com).
- This *C++ How to Program: Third Edition Instructor's Manual* on CD contains answers to most of the exercises in the textbook. The programs are separated into directories by chapter and exercise number.
- The optional *C++ Multimedia Cyber Classroom: Third Edition* is an interactive multimedia CD version of the book for Windows. Its features include audio walk-throughs of programs, section review questions (which are available only on the *C++ Multimedia Cyber Classroom: Third Edition*), a text-search engine, the ability to execute example programs and more. The *Cyber Classroom* helps students get more out of their courses. The *Cyber Classroom* is also useful for students who miss a lecture and have to catch up quickly. The *Cyber Classroom* is available as a stand-alone product (see the last few pages of the textbook for the ISBN number) or bundled with the textbook (at a discount) in a product called *The Complete C++ Training Course: Third Edition* (ISBN# 0-13-089563-6). We discuss the *Cyber Classroom* in further detail later in the Preface.
- *Companion Web site* (www.prenhall.com/deitel) provides instructor and student resources. Instructor resources include textbook appendices (e.g., Appendix D, "C++ Internet and Web Resources") and a syllabus manager for lesson planning. Student resources include chapter objectives, true/false questions, chapter highlights, reference materials and a message board.
- Customizable *PowerPoint® Instructor Lecture Notes*, with many complete features including source code and key discussion points for each program and major

illustration. These lecture notes are available for instructors and students at no charge at our Web site www.deitel.com.

- *Lab Manual* (available Spring 2001)—a for-sale item containing closed-lab sessions.

A Revolution in Software Development

For years, hardware has been improving dramatically, but software, for some reason, seemed to resist almost every attempt to build it faster and to build it better. Today we are in the middle of a revolution in the way software is being designed and written. That revolution is based on the common-sense, hardware notion of standardized, interchangeable parts, exactly as used by Henry Ford in the days of the Model T Ford. These software components are called “objects”—more properly, “classes,” which are the “cookie cutters” out of which objects are produced.

The most mature of the well-known object-oriented languages is Smalltalk, developed in the early 1970s at Xerox’s Palo Alto Research Center. But the most widely used object-oriented language—by a factor of 10 over Smalltalk—is C++ developed by Bjarne Stroustrup and others in the early 1980s at AT&T. In the time between the publication of the first and second editions of this book, another contender appeared on the scene—the Java object-oriented programming language, developed in the early 1990s by James Gosling and others at Sun Microsystems.

Why a major new object-oriented programming language every 10 years? Smalltalk was truly ahead of its time as a research experiment. C++ was right for its time and for today’s high-performance systems programming and applications development needs. Java™ offered developers the ability to create highly portable multimedia-intensive and networking-intensive Internet/World Wide Web-based applications.

Procedural Programming, Object-Based Programming, Object-Oriented Programming and Generic Programming

In this book, you will master the five key components of C++ as well as four contemporary programming paradigms:

1. *C procedural programming*—Chapters 1–5 and 16–18; key topics include data types, control structures, functions, arrays, pointers, strings, structures, bit manipulation, character manipulation, preprocessing and others.
2. *C++ procedural programming enhancements to C*—Sections 3.15–3.21; key topics include **inline** functions, references, default arguments, function overloading and function templates.
3. *C++ object-based programming*—Chapters 6–8; key topics include abstract data types, classes, objects, encapsulation, information hiding, member access control, constructors, destructors, software reusability, constant objects and member functions, composition, friendship, dynamic memory allocation, **static** members, **this** pointer and others.

4. C++ *object-oriented programming*—Chapters 9–15, 19 and 21; key topics include base classes, single inheritance, derived classes, multiple inheritance, **virtual** functions, dynamic binding, polymorphism, pure **virtual** functions, abstract classes, concrete classes, stream input/output, class templates, exception handling, file processing, data structures, strings as full-fledged objects, data type **bool**, cast operators, namespaces, run-time type information (RTTI), **explicit** constructors and **mutable** members.
5. C++ *generic programming*—Chapter 20—the largest chapter in the book; key topics include the Standard Template Library (STL), templated containers, sequence containers, associative containers, container adaptors, iterators that traverse templated containers and algorithms that process the elements of templated containers.

Evolving from Pascal and C to C++ and Java™

C++ has replaced C as the systems implementation language of choice in industry. But C programming will continue to be an important and valuable skill because of the enormous amount of C legacy code that must be maintained. Dr. Harvey M. Deitel has been teaching introductory programming courses in college environments for two decades with an emphasis on developing clearly written, well-structured programs. Much of what is taught in these courses is the basic principles of programming with an emphasis on the effective use of control structures and functionalization. We have presented this material exactly the way HMD has done in his college courses. There are some pitfalls, but where these occur, we point them out and explain procedures for dealing with them effectively. Our experience has been that students handle the course in about the same manner as they handle introductory Pascal or C courses. There is one noticeable difference though: Students are highly motivated by the fact that they are learning a leading-edge language (C++) and a leading-edge programming paradigm (object-oriented programming) that will be immediately useful to them as they leave the college environment. This increases their enthusiasm for the material—a big help when you consider that C++ is more difficult to learn than Pascal or C.

Our goal was clear: Produce a C++ programming textbook for introductory college-level courses in computer programming for students with little or no programming experience, yet offer the depth and the rigorous treatment of theory and practice demanded by traditional, upper-level C++ courses. To meet these goals, we produced a book larger than other C++ texts—this because our text also patiently teaches the principles of procedural programming, object-based programming, object-oriented programming and generic programming. Hundreds of thousands of people have studied this material in academic courses and professional seminars worldwide.

Until the early 1990s, computer science courses were focused on procedural programming in Pascal and C. Since then, these courses have largely switched to object-oriented programming in C++ and Java. At Deitel & Associates, Inc., we are focussed on producing quality educational materials for today's leading-edge programming languages. As *C++ How to Program: Third Edition* goes to the presses, we are working on *Java How to Program: Fourth Edition*, *Advanced C++ How to Program* and *Advanced Java How to Program*.

Introducing Object Orientation from Chapter 1!

We faced a difficult challenge in designing this book. Should the book present a pure object-oriented approach? Or should it present a hybrid approach balancing procedural programming with object-oriented programming?

Many instructors who will teach from this text have been teaching procedural programming (probably in C or Pascal). C++ itself is not a purely object-oriented language. Rather it is a hybrid language that enables both procedural programming and object-oriented programming.

So we chose the following approach. The first five chapters of the book introduce procedural programming in C++. They present computer concepts, control structures, functions, arrays, pointers and strings. These chapters cover the C portion of C++ and the C++ “procedural enhancements” to C.

We have done something to make these first five chapters really unique. At the end of each of these chapters, we have included a special section entitled, “Thinking About Objects.” These sections introduce the concepts and terminology of object orientation to help students begin familiarizing themselves with what objects are and how they behave.

The Chapter 1 “Thinking About Objects” section introduces the concepts and terminology of object orientation. The sections in Chapters 2 through 5 present a requirements specification for a substantial object-oriented system project, namely building an elevator simulator and carefully guide the student through the typical phases of the object-oriented design process. These sections discuss how to identify the objects in a problem, how to specify the objects’ attributes and behaviors, and how to specify the interactions among objects. By the time the student has finished Chapter 5, he or she has completed a careful object-oriented design of the elevator simulator and is ready—if not eager—to begin programming the elevator in C++. Chapters 6 and 7 cover data abstraction and classes. These chapters also contain “Thinking About Objects” sections that ease students through the various stages of programming their elevator simulators in C++. Chapter 9’s “Thinking About Objects” section applies C++ inheritance concepts to the elevator simulator.

About this Book

C++ *How to Program* contains a rich collection of examples, exercises and projects drawn from many fields to provide the student with a chance to solve interesting real-world problems. The book concentrates on the principles of good software engineering and stresses program clarity. We avoid arcane terminology and syntax specifications in favor of teaching by example.

This book is written by educators who spend most of their time teaching and writing about edge-of-the-practice programming languages.

The text places a strong emphasis on pedagogy. For example, virtually every new concept of either C++ or object-oriented programming is presented in the context of a complete, working C++ program immediately followed by a window showing the program’s output. Reading these programs is much like entering and running them on a computer. We call this our “live-code approach.”

Among the other pedagogical devices in the text are a set of *Objectives* and an *Outline* at the beginning of every chapter; *Common Programming Errors*, *Good Programming Practices*, *Performance Tips*, *Portability Tips*, *Software Engineering Observations* and

Testing and Debugging Tips enumerated in, and summarized at, the end of each chapter; comprehensive bullet-list-style *Summary* and alphabetized *Terminology* sections in each chapter; *Self-Review Exercises and Answers* in each chapter; and the richest collection of *Exercises* in any C++ book.

The exercises range from simple recall questions to lengthy programming problems to major projects. Instructors requiring substantial term projects will find many appropriate problems listed in the exercises for Chapters 3 through 21. We have put a great deal of effort into the exercises to enhance the value of this course for the student.

In writing this book, we have used a variety of C++ compilers. For the most part, the programs in the text will work on all ANSI/ISO compilers.

This text is based on the C++ programming language as developed by Accredited Standards Committee X3, Information Technology and its Technical Committee X3J16, Programming Language C++, respectively. This language was approved by the International Standards Organization (ISO). For further details, contact:

X3 Secretariat
1250 Eye Street NW
Washington DC 20005

The serious programmer should read these documents carefully and reference them regularly. These documents are not tutorials. Rather they define C++ and C with the extraordinary level of precision that compiler implementors and “heavy-duty” developers demand.

We have carefully audited our presentation against these documents. Our book is intended to be used at the introductory and intermediate levels. We have not attempted to cover every feature discussed in these comprehensive documents.

Objectives

Each chapter begins with a statement of objectives. This tells the student what to expect and gives the student an opportunity, after reading the chapter, to determine if he or she has met these objectives. It is a confidence builder and a source of positive reinforcement.

Quotations

The learning objectives are followed by a series of quotations. Some are humorous, some are philosophical and some offer interesting insights. Our students enjoy relating the quotations to the chapter material. You may appreciate some of the quotations more *after* reading the chapters.

Outline

The chapter outline helps the student approach the material in top-down fashion. This, too, helps students anticipate what is to come and set a comfortable and effective learning pace.

Sections

Each chapter is organized into small sections that address key C++ topics.

13,741 Lines of Syntax-Colored Code in 268 Example Programs (with Program Outputs)

We present C++ features in the context of complete, working C++ programs; each program is immediately followed by a window containing the outputs produced when the program

is run—we call this our “live-code approach.” This enables the student to confirm that the programs run as expected. Relating outputs back to the program statements that produce those outputs is an excellent way to learn and to reinforce concepts. Our programs exercise the diverse features of C++. Reading the book carefully is much like entering and running these programs on a computer. The code is “syntax colored” with C++ keywords, comments and other program text each appearing in different colors. This makes it much easier to read the code—students will especially appreciate the syntax coloring when they read the many more substantial programs we present.

469 Illustrations/Figures

An abundance of colorized charts and line drawings is included. The discussion of control structures in Chapter 2 features carefully drawn flowcharts. (Note: We do not teach the use of flowcharting as a program development tool, but we do use a brief flowchart-oriented presentation to specify the precise operation of C++’s control structures.) Chapter 15, “Data Structures,” uses colorized line drawings to illustrate the creation and maintenance of linked lists, queues, stacks and binary trees. The remainder of the book is abundantly illustrated.

625 Programming Tips

We have included six design elements to help students focus on important aspects of program development, testing and debugging, performance and portability. We highlight hundreds of these tips in the form of *Good Programming Practices*, *Common Programming Errors*, *Performance Tips*, *Portability Tips*, *Software Engineering Observations* and *Testing and Debugging Tips*. These tips and practices represent the best we have been able to glean from almost six decades (combined) of programming and teaching experience. One of our students—a mathematics major—told us recently that she feels this approach is somewhat like the highlighting of axioms, theorems and corollaries in mathematics books; it provides a basis on which to build good software.



112 Good Programming Practices

Good Programming Practices are highlighted in the text. They call the student’s attention to techniques that help produce better programs. When we teach introductory courses to nonprogrammers, we state that the “buzzword” of each course is “clarity,” and we tell the students that we will highlight (in these *Good Programming Practices*) techniques for writing programs that are clearer, more understandable and more maintainable.



216 Common Programming Errors

Students learning a language—especially in their first programming course—tend to make certain kinds of errors frequently. Focusing on these *Common Programming Errors* helps students avoid making the same errors. It also helps reduce long lines outside instructors’ offices during office hours!



87 Performance Tips

In our experience, teaching students to write clear and understandable programs is by far the most important goal for a first programming course. But students want to write the programs that run the fastest, use the least memory, require the smallest number of keystrokes, or dazzle in other nifty ways. Students really care about per-

formance. They want to know what they can do to “turbo charge” their programs. So we have include *Performance Tips* to highlight opportunities for improving program performance.



37 Portability Tips

Software development is a complex and expensive activity. Organizations that develop software must often produce versions customized to a variety of computers and operating systems. So there is a strong emphasis today on portability, i.e., on producing software that will run on a variety of computer systems with few, if any, changes. Many people tout C++ as an appropriate language for developing portable software, especially because of C++’s close relationship to ANSI/ISO C and the fact that ANSI/ISO C++ is the global C++ standard. Some people assume that if they implement an application in C++, the application will automatically be portable. This is simply not the case. Achieving portability requires careful and cautious design. There are many pitfalls. We include numerous *Portability Tips* to help students write portable code.



146 Software Engineering Observations

The object-oriented programming paradigm requires a complete rethinking about the way we build software systems. C++ is an effective language for performing good software engineering. The *Software Engineering Observations* highlight techniques, architectural issues and design issues, etc. that affect the architecture and construction of software systems, especially large-scale systems. Much of what the student learns here will be useful in upper-level courses and in industry as the student begins to work with large, complex real-world systems.



27 Testing and Debugging Tips

This “tip type” may be misnamed. When we decided to incorporate *Testing and Debugging Tips* into this new edition, we thought these tips would be suggestions for testing programs to expose bugs and suggestions for removing those bugs. In fact, most of these tips tend to be observations about capabilities and features of C++ that prevent bugs from getting into programs in the first place.

Summary

Each chapter ends with additional pedagogical devices. We present an extensive, bullet-list-style *Summary* in every chapter. This helps the student review and reinforce key concepts. There is an average of 37 summary bullets per chapter.

Terminology

We include a *Terminology* section with an alphabetized list of the important terms defined in the chapter—again, further reinforcement. There is an average of 72 terms per chapter.

Summary of Tips, Practices and Errors

We collect and list from the chapter the *Good Programming Practices*, *Common Programming Errors*, *Performance Tips*, *Portability Tips*, *Software Engineering Observations* and *Testing and Debugging Tips*.

554 Self-Review Exercises and Answers (Count Includes Separate Parts)

Extensive *Self-Review Exercises* and *Answers to Self-Review Exercises* are included for self study. This gives the student a chance to build confidence with the material and prepare to attempt the regular exercises.

877 Exercises (Count Includes Separate Parts; 1431 Total Exercises)

Each chapter concludes with a substantial set of exercises including simple recall of important terminology and concepts; writing individual C++ statements; writing small portions of C++ functions and classes; writing complete C++ functions, classes and programs; and writing major term projects. The large number of exercises enables instructors to tailor their courses to the unique needs of their audiences and to vary course assignments each semester. Instructors can use these exercises to form homework assignments, short quizzes and major examinations.

550-page Instructor's Manual with Solutions to the Exercises

The solutions for the exercises are included on the *Instructor's CD* and on the disks *available only to instructors* through their Prentice Hall representatives. **[NOTE: Please do not write to us requesting the instructor's CD. Distribution of this CD is limited strictly to college professors teaching from the book. Instructors may obtain the solutions manual only from their Prentice Hall representatives.]** Solutions to approximately half of the exercises are included on the *C++ Multimedia Cyber Classroom: Third Edition CD* (available September 2000; please see the last few pages of this book for ordering instructions).

4523 Index Entries (Total of 7653 Counting Multiple References)

We have included an extensive *Index* at the back of the book. This helps the student find any term or concept by keyword. The *Index* is useful to people reading the book for the first time and is especially useful to practicing programmers who use the book as a reference. Most of the terms in the *Terminology* sections appear in the *Index* (along with many more index items from each chapter). Thus, the student can use the *Index* in conjunction with the *Terminology* sections to be sure he or she has covered the key material of each chapter.

A Tour of the Book

The book is divided into several major parts. The first part, Chapters 1 through 5, presents a thorough treatment of procedural programming in C++ including data types, input/output, control structures, functions, arrays, pointers and strings. The “Thinking About Objects” section at the ends of Chapters 1–5 introduce object technology, present an interesting and challenging optional case study in designing and implementing a substantial object-oriented system.

The second part, Chapters 6 through 8, presents a substantial treatment of data abstraction with classes, objects and operator overloading. This section might effectively be called, “Programming with Objects.” The “Thinking About Objects” sections at the ends of Chapters 6 and 7 develop and present a 1000-line C++ program that implements the design presented in Chapters 2–5.

The third part, Chapters 9 and 10, presents inheritance, virtual functions and polymorphism—the root technologies of true object-oriented programming.

The “Thinking About Objects” section at the end of Chapter 9 incorporates inheritance into the design and implementation of the elevator simulator.

The fourth part, chapters 11 and 14, presents C++-style stream-oriented input/output including using stream I/O with the keyboard, the screen, files and character arrays; both sequential file processing and direct-access (i.e., random access) file processing are discussed.

The fifth part, Chapters 12 and 13, discusses two of the more recent additions to C++, namely templates and exception handling. Templates, also called parameterized types, encourage software reusability. Exceptions help programmers develop more robust, fault-tolerant, business-critical and mission-critical systems.

The sixth part, Chapter 15, presents a thorough treatment of dynamic data structures such as linked lists, queues, stacks and trees. This chapter, when supplemented with the treatment of the Standard Template Library (STL) in Chapter 20, creates a rich treatment of data structures that makes a nice C++ supplement to traditional computer science data structures and algorithms courses.

The seventh part, Chapters 16 through 18 discuss a variety of topics including bit, character and string manipulation; the preprocessor and miscellaneous “Other Topics.”

The last part of the main text, Chapters 19 through 21, is devoted to the latest enhancements to C++ and to the C++ Standard Library which have been included in the ANSI/ISO C++ Standard. These include discussions of class `string`, string stream processing, the Standard Template Library and a potpourri of other recent additions to C++.

The end matter of the book consists of reference materials that support the main text including Appendices on operator precedence, the ASCII character set, number systems (binary, decimal, octal and hexadecimal) and C++ Internet/World Wide Web resources. A bibliography is included to encourage further reading. The text concludes with a detailed index that helps the reader locate any terms in the text by keyword. Now let us look at each of the chapters in detail.

Chapter 1—Introduction to Computers and C++ Programming—discusses what computers are, how they work and how they are programmed. It introduces the notion of structured programming and explains why this set of techniques has fostered a revolution in the way programs are written. The chapter gives a brief history of the development of programming languages from machine languages, to assembly languages, to high-level languages. The origin of the C++ programming language is discussed. The chapter includes an introduction to a typical C++ programming environment and gives a concise introduction to writing C++ programs. A detailed treatment of decision making and arithmetic operations in C++ is presented. We have introduced a new, more open, easier to read “look and feel” for our C++ source programs, most notably using syntax coloring to highlight keyword comments and regular program text; and to make programs more readable. After studying this chapter, the student will understand how to write simple, but complete, C++ programs. We discuss the explosion in interest in the Internet that has occurred with the advent of the World Wide Web and the Java programming language. We discuss `namespaces` and the `using` statement for the benefit of readers with access to standard-compliant compilers. We use the new-style header files. It will take a few years to “clear out” the older compilers that are still widely used. Readers plunge right in with object-orientation in the “Thinking About Objects” section which introduces the basic terminology of object technology.

Chapter 2—Control Structures—introduces the notion of algorithms (procedures) for solving problems. It explains the importance of using control structures effectively in producing programs that are understandable, debuggable, maintainable and more likely to work properly on the first try. It introduces the sequence structure, selection structures (**if**, **if/else** and **switch**) and repetition structures (**while**, **do/while** and **for**). It examines repetition in detail and compares counter-controlled loops and sentinel-controlled loops. It explains the technique of top-down, stepwise refinement that is critical to the production of properly structured programs and presents the popular program design aid, pseudocode. The methods and approaches used in Chapter 2 are applicable to effective use of control structures in any programming language, not just C++. This chapter helps the student develop good programming habits in preparation for dealing with the more substantial programming tasks in the remainder of the text. The chapter concludes with a discussion of logical operators—**&&** (and), **||** (or) and **!** (not). The keyword table was enhanced with the new C++ keywords introduced in the ANSI/ISO C++ Standard. We introduce the new-style **static_cast** operator. This is safer than using the old-style casting C++ inherited from C. We added the “Peter Minuit” exercise so students can see the wonders of compound interest—with the computer doing most of the work! We discuss the new scoping rules for loop counters in **for**-loops. In the “Thinking About Objects” section, we begin the first phase of an object-oriented design (OOD) for the elevator simulator—identifying the classes needed to implement the simulator. We also introduce the UML use case, class and object diagrams and the concepts of associations, multiplicity, composition, roles and links.

Chapter 3—Functions—discusses the design and construction of program modules. C++’s function-related capabilities include standard-library functions, programmer-defined functions, recursion, call-by-value and call-by-reference capabilities. The techniques presented in Chapter 3 are essential to the production of properly structured programs, especially the kinds of larger programs and software that system programmers and application programmers are likely to develop in real-world applications. The “divide and conquer” strategy is presented as an effective means for solving complex problems by dividing them into simpler interacting components. Students enjoy the treatment of random numbers and simulation, and they appreciate the discussion of the dice game of craps which makes elegant use of control structures. The chapter offers a solid introduction to recursion and includes a table summarizing the dozens of recursion examples and exercises distributed throughout the remainder of the book. Some texts leave recursion for a chapter late in the book; we feel this topic is best covered gradually throughout the text. The extensive collection of 60 exercises at the end of the chapter includes several classical recursion problems such as the Towers of Hanoi. The chapter discusses the so-called “C++ enhancements to C,” including **inline** functions, reference parameters, default arguments, the unary scope resolution operator, function overloading and function templates. The header files table has been modified to include many of the new header files that the reader will use throughout the book. Please do Exercise 3.54 on adding a wagering capability to the craps program. In “Thinking About Objects” section, we determine many of the class attributes needed to implement the elevator simulator. We also introduce the UML statechart and activity diagrams and the concepts of events and actions as they relate to these diagrams.

Chapter 4—Arrays—discusses the structuring of data into arrays, or groups, of related data items of the same type. The chapter presents numerous examples of both

single-subscripted arrays and double-subscripted arrays. It is widely recognized that structuring data properly is just as important as using control structures effectively in the development of properly structured programs. Examples in the chapter investigate various common array manipulations, printing histograms, sorting data, passing arrays to functions and an introduction to the field of survey data analysis (with simple statistics). A feature of this chapter is the discussion of elementary sorting and searching techniques and the presentation of binary searching as a dramatic improvement over linear searching. The 94 end-of-chapter exercises include a variety of interesting and challenging problems such as improved sorting techniques, the design of an airline reservations system, an introduction to the concept of turtle graphics (made famous in the LOGO language) and the Knight's Tour and Eight Queens problems that introduce the notion of heuristic programming so widely employed in the field of artificial intelligence. The exercises conclude with many recursion problems including the selection sort, palindromes, linear search, binary search, the Eight Queens, printing an array, printing a string backwards and finding the minimum value in an array. This chapter still uses C-style arrays which, as you will see in Chapter 5, are really pointers to the array contents in memory. We are certainly committed to arrays as full-fledged objects. In Chapter 8, we use the techniques of operator overloading to craft a valuable **Array** class out of which we create **Array** objects that are much more robust and pleasant to program with than the arrays of Chapter 4. In Chapter 20, "Standard Template Library (STL)," we introduce STL's class **vector** which, when used with the iterators and algorithms discussed in Chapter 20, creates a solid treatment of arrays as full-fledged objects. In the "Thinking About Objects" section, we determine many of the operations (behaviors) of the classes in the elevator simulation. We also introduce the UML sequence diagram and the concept of messages sent between objects.

Chapter 5—Pointers and Strings—presents one of the most powerful and difficult-to-master features of the C++ language, namely pointers. The chapter provides detailed explanations of pointer operators, call by reference, pointer expressions, pointer arithmetic, the relationship between pointers and arrays, arrays of pointers and pointers to functions. There is an intimate relationship between pointers, arrays and strings in C++, so we introduce basic string-manipulation concepts and include a discussion of some of the most popular string-handling functions, namely **getline** (input a line of text), **strcpy** and **strncpy** (copy a string), **strcat** and **strncat**, (concatenate two strings) **strcmp** and **strncmp** (compare two strings), **strtok** ("tokenize" a string into its pieces) and **strlen** (compute the length of a string). The 49 chapter exercises include a simulation of the classic race between the tortoise and the hare, card shuffling and dealing algorithms, recursive quicksort and recursive maze traversals. A special section entitled "Building Your Own Computer" is also included. This section explains machine-language programming and proceeds with a project involving the design and implementation of a computer simulator that allows the reader to write and run machine language programs. This unique feature of the text will be especially useful to the reader who wants to understand how computers really work. Our students enjoy this project and often implement substantial enhancements, many of which are suggested in the exercises. In Chapter 15, another special section guides the reader through building a compiler; the machine language produced by the compiler is then executed on the machine language simulator produced in Chapter 7. Information is communicated from the compiler to the simulator in sequential files which we discuss in Chapter 14. A second special section includes challenging string-manipulation exercises

related to text analysis, word processing, printing dates in various formats, check protection, writing the word equivalent of a check amount, Morse Code and metric-to-English conversions. The reader will want to revisit these string-manipulation exercises after studying class `string` in Chapter 19. Many people find that the topic of pointers is, by far, the most difficult part of an introductory programming course. In C and “raw C++” arrays and strings are really pointers to array and string contents in memory. Even function names are pointers. Studying this chapter carefully should reward you with a deep understanding of the complex topic of pointers. Again, we cover arrays and strings as full-fledged objects later in the book. In Chapter 8, we use operator overloading to craft customized `Array` and `String` classes. In Chapter 19, we discuss Standard Library class `string` and show how to manipulate `string` objects. In Chapter 20 we discuss class `vector` for implementing array objects. Chapter 5 is loaded with challenging exercises. Please be sure to try the *Special Section: Building Your Own Computer*. In the “Thinking About Objects” section, we determine many of the collaborations (interactions between objects in the system) needed to implement the elevator system and represent these collaborations using the UML collaboration diagram. We also include a bibliography and a list of Internet and World Wide Web resources that contain the UML 1.3 specifications and other reference materials, general resources, tutorials, FAQs, articles, whitepapers and software.

Chapter 6—Classes and Data Abstraction—begins our discussion of object-based programming. The chapter represents a wonderful opportunity for teaching data abstraction the “right way”—through a language (C++) expressly devoted to implementing abstract data types (ADTs). In recent years, data abstraction has become a major topic in introductory computing courses. Chapters 6 through 8 include a solid treatment of data abstraction. Chapter 6 discusses implementing ADTs as `structs`, implementing ADTs as C++-style `classes` and why this approach is superior to using `structs`, accessing `class` members, separating interface from implementation, using access functions and utility functions, initializing objects with constructors, destroying objects with destructors, assignment by default memberwise copy and software reusability. The chapter exercises challenge the student to develop classes for complex numbers, rational numbers, times, dates, rectangles, huge integers and for playing tic-tac-toe. Students generally enjoy game-playing programs. The “Thinking About Objects” section asks you to write a class header file for each of the classes in your elevator simulator. The more mathematically inclined reader will enjoy the exercises on creating class `Complex` (for complex numbers), class `Rational` (for rational numbers) and class `HugeInteger` (for arbitrarily large integers). In the “Thinking About Objects” section, we use the UML class diagram developed in previous sections to outline the C++ header files that define our classes. We also introduce the concept of handles to objects in the system and we begin to study how to implement handles in C++.

Chapter 7—Classes Part II—continues the study of classes and data abstraction. The chapter discusses declaring and using constant objects, constant member functions, composition—the process of building classes that have objects of other classes as members, `friend` functions and `friend` classes that have special access rights to the `private` and `protected` members of classes, the `this` pointer that enables an object to know its own address, dynamic memory allocation, `static` class members for containing and manipulating class-wide data, examples of popular abstract data types (arrays, strings and queues), container classes and iterators. The chapter exercises ask the student to develop a savings

account class and a class for holding sets of integers. In our discussion of **const** objects, we briefly mention keyword **mutable** which, as we will see in Chapter 21, is used in a subtle manner to enable modification of “non-visible” implementation in **const** objects. We discuss dynamic memory allocation with **new** and **delete**. When **new** fails, it returns a 0 pointer in pre-standard C++. We use this pre-standard style in Chapters 7-12. We defer to Chapter 13 the discussion of the new style of **new** failure in which **new** now “throws an exception.” We motivate the discussion of **static** class members with a video-game-based example. We emphasize throughout the book and in our professional seminars how important it is to hide implementation details from clients of a class; then, we show **private** data on our class headers, which certainly reveals implementation. We discuss proxy classes, a nice means of hiding even **private** data from clients of a class. The “Thinking About Objects” section asks you to incorporate dynamic memory management and composition into your elevator simulator. Students will enjoy the exercise creating class **IntegerSet**. This serves as excellent motivation for the treatment of operator overloading in Chapter 8. In the “Thinking About Objects” section, we present a complete elevator simulator C++ program (approximately 1000 lines of code) and a detailed code walkthrough. The code follows directly from the UML-based design created in previous sections and employs our good programming practices, including the use of **static** and **const** data members and functions. We also discuss dynamic-memory allocation, composition and object interaction via handles and how to use forward declarations to avoid the circular include problem.

Chapter 8—Operator Overloading—as one of the most popular topics in our C++ courses. Students really enjoy this material. They find it a perfect match with the discussion of abstract data types in Chapters 6 and 7. Operator overloading enables the programmer to tell the compiler how to use existing operators with objects of new types. C++ already knows how to use these operators with objects of built-in types such as integers, floats and characters. But suppose we create a new string class—what would the plus sign mean when used between string objects? Many programmers use plus with strings to mean concatenation. In Chapter 8, the programmer will learn how to “overload” the plus sign so that when it is written between two string objects in an expression, the compiler will generate a function call to an “operator function” that will concatenate the two strings. The chapter discusses the fundamentals of operator overloading, restrictions in operator overloading, overloading with class member functions vs. with nonmember functions, overloading unary and binary operators and converting between types. A feature of the chapter is the collection of substantial case studies including an array class, a string class, a date class, a huge integer class and a complex numbers class (the last two appear with full source code in the exercises). The more mathematically inclined student will enjoy creating the polynomial class in the exercises. This material is different from what you do in most programming languages and courses. Operator overloading is a complex topic, but an enriching one. Using operator overloading wisely helps you add that extra “polish” to your classes. The discussions of class **Array** and class **String** are particularly valuable to students who will go on to use the Standard Library classes **string** and **vector**. With the techniques of Chapters 6, 7 and 8, it is possible to craft a **Date** class that, if we had been using it for the last two decades, could easily have eliminated a major portion of the so-called “Year 2000 (or Y2K) Problem.” The exercises encourage the student to add operator overloading to classes **Complex**, **Rational** and **HugeInteger** to enable convenient manipulation of

objects of these classes with operator symbols—as in mathematics—rather than with function calls as the student did in the Chapter 7 exercises.

Chapter 9—Inheritance—deals with one of the most fundamental capabilities of object-oriented programming languages. Inheritance is a form of software reusability in which new classes are developed quickly and easily by absorbing the capabilities of existing classes and adding appropriate new capabilities. The chapter discusses the notions of base classes and derived classes, **protected** members, **public** inheritance, **protected** inheritance, **private** inheritance, direct base classes, indirect base classes, constructors and destructors in base classes and derived classes and software engineering with inheritance. The chapter compares inheritance (“is a” relationships) with composition (“has a” relationships) and introduces “uses a” and “knows a” relationships. A feature of the chapter is its several substantial case studies. In particular, a lengthy case study implements a point, circle, cylinder class hierarchy. The chapter concludes with a case study on multiple inheritance—an advanced feature of C++ that enables a derived class to be formed by inheriting attributes and behaviors from several base classes. The exercises ask the student to compare the creation of new classes by inheritance vs. composition; to extend the various inheritance hierarchies discussed in the chapter; to write an inheritance hierarchy for quadrilaterals, trapezoids, parallelograms, rectangles and squares; and to create a more general shape hierarchy with two-dimensional shapes and three-dimensional shapes. We modify our inheritance hierarchy for university community members to show a nice example of multiple inheritance. In Chapter 21 we continue our discussion of multiple inheritance by exposing the problems caused by so-called “diamond inheritance” and showing how to solve these problems with **virtual** base classes. In the “Thinking About Objects” section, we update the elevator simulation design and implementation to incorporate inheritance. We also suggest further modifications so that the student may then design and implement, using the tools presented in the previous sections.

Chapter 10—Virtual Functions and Polymorphism—deals with another of the fundamental capabilities of object-oriented programming, namely polymorphic behavior. When many classes are related through inheritance to a common base class, each derived-class object may be treated as a base-class object. This enables programs to be written in a general manner independent of the specific types of the derived-class objects. New kinds of objects can be handled by the same program, thus making systems more extensible. Polymorphism enables programs to eliminate complex **switch** logic in favor of simpler “straight-line” logic. A screen manager of a video game, for example, can simply send a draw message to every object in a linked list of objects to be drawn. Each object knows how to draw itself. A new object can be added to the program without modifying that program as long as that new object also knows how to draw itself. This style of programming is typically used to implement today’s popular graphical user interfaces (GUIs). The chapter discusses the mechanics of achieving polymorphic behavior through the use of **virtual** functions. It distinguishes between abstract classes (from which objects cannot be instantiated) and concrete classes (from which objects can be instantiated). Abstract classes are useful for providing an inheritable interface to classes throughout the hierarchy. A feature of the chapter is its two major polymorphism case studies—a payroll system and another version of the point, circle, cylinder shape hierarchy discussed in Chapter 9. The chapter exercises ask the student to discuss a number of conceptual issues and approaches, add

abstract classes to the shape hierarchy, develop a basic graphics package, modify the chapter's employee class—and pursue all these projects with **virtual** functions and polymorphic programming. The chapter's two polymorphism case studies show a contrast in inheritance styles. The first example (of a payroll system) is a clear, “sensible” use of inheritance. The second, which builds on the point, circle, cylinder hierarchy developed in Chapter 9, is an example of what some professionals call “structural inheritance”—not as natural and sensible as the first, but “mechanically correct” nevertheless. We use this second example because of the section entitled “Polymorphism, **virtual** Functions and Dynamic Binding “Under the Hood.” We deliver C++ professional seminars to senior software engineers. These people appreciated the two polymorphism examples in the first edition, but they felt something was missing from our presentations. Yes, they said, we showed them how to program with polymorphism in C++. But they wanted more. They told us they were concerned about the operating overhead of programming polymorphically. It is a nice feature, they said, but it clearly has costs. So our professional audiences insisted that we provide a deeper explanation that showed precisely how polymorphism is implemented in C++, and hence, precisely what execution time and memory “costs” one must pay when programming with this powerful capability. We responded by developing an illustration that shows the *vtables* (**virtual** function tables) that the C++ compiler automatically builds to support the polymorphic programming style. We drew these tables in our classes in which we discussed the point, circle, cylinder shape hierarchy. Our audiences indicated that this indeed gave them the information to decide whether polymorphism was an appropriate programming style for each new project they would tackle. We have included this presentation in Section 10.10 and the *vtable* illustration in Fig. 10.2. Please study this presentation carefully. It will give you a much deeper understanding of what is really occurring in the computer when you program with inheritance and polymorphism.

Chapter 11—C++ Stream Input/Output—contains a comprehensive treatment of C++ object-oriented input/output. The chapter discusses the various I/O capabilities of C++ including output with the stream insertion operator, input with the stream extraction operator, type-safe I/O (a nice improvement over C), formatted I/O, unformatted I/O (for performance), stream manipulators for controlling the stream base (decimal, octal, or hexadecimal), floating-point numbers, controlling field widths, user-defined manipulators, stream format states, stream error states, I/O of objects of user-defined types and tying output streams to input streams (to ensure that prompts actually appear before the user is expected to enter responses). The extensive exercise set asks the student to write various programs that test most of the I/O capabilities discussed in the text.

Chapter 12—Templates—discusses one of the more recent additions to C++. Function templates were introduced in Chapter 3. Chapter 12 presents an additional function template example. Class templates enable the programmer to capture the essence of an abstract data type (such as a stack, an array, or a queue) and then create—with minimal additional code—versions of that ADT for particular types (such as a queue of **int**, a queue of **float**, a queue of strings, etc.). For this reason, template classes are often called parameterized types. The chapter discusses using type parameters and nontype parameters and considers the interaction among templates and other C++ concepts, such as inheritance, **friends** and **static** members. The exercises challenge the student to write a variety of function templates and class templates, and to employ these in complete programs. We

greatly enhance the treatment of templates with the discussion of the Standard Template Library (STL) containers, iterators and algorithms in Chapter 20.

Chapter 13—Exception Handling—discusses one of the more recent enhancements to the C++ language. Exception handling enables the programmer to write programs that are more robust, more fault tolerant and more appropriate for business-critical and mission-critical environments. The chapter discusses when exception handling is appropriate; introduces the basics of exception handling with **try** blocks, **throw** statements and **catch** blocks; indicates how and when to rethrow an exception; explains how to write an exception specification and process unexpected exceptions; and discusses the important ties between exceptions and constructors, destructors and inheritance. A feature of the chapter is its 43 exercises that walk the student through implementing programs that illustrate the diversity and power of C++'s exception handling capabilities. We discuss rethrowing an exception and we illustrate both ways **new** can fail when memory is exhausted. Prior to the C++ draft standard **new** failed by returning 0, much as **malloc** fails in C by returning a **NULL** pointer value. We show the new style of **new** failing by throwing a **bad_alloc** (bad allocation) exception. We illustrate how to use **set_new_handler** to specify a custom function to be called to deal with memory exhaustion situations. We discuss the **auto_ptr** class template to guarantee that dynamically allocated memory will be properly **deleted** to avoid memory leaks. We present the new Standard Library exception hierarchy.

Chapter 14—File Processing—discusses the techniques used to process text files with sequential access and random access. The chapter begins with an introduction to the data hierarchy from bits, to bytes, to fields, to records, to files. Next, C++'s simple view of files and streams is presented. Sequential-access files are discussed using programs that show how to open and close files, how to store data sequentially in a file and how to read data sequentially from a file. Random-access files are discussed using programs that show how to sequentially create a file for random access, how to read and write data to a file with random access and how to read data sequentially from a randomly accessed file. The fourth random-access program combines many of the techniques of accessing files both sequentially and randomly into a complete transaction-processing program. Students in our industry seminars have told us that after studying the material on file processing, they were able to produce substantial file-processing programs that were immediately useful in their organizations. The exercises ask the student to implement a variety of programs that build and process both sequential-access files and random-access files. The closely related material on string stream processing has been positioned at the end of Chapter 19.

Chapter 15—Data Structures—discusses the techniques used to create and manipulate dynamic data structures. The chapter begins with discussions of self-referential classes and dynamic memory allocation and proceeds with a discussion of how to create and maintain various dynamic data structures including linked lists, queues (or waiting lines), stacks and trees. For each type of data structure, we present complete, working programs and show sample outputs. The chapter really helps the student master pointers. The chapter includes abundant examples using indirection and double indirection—a particularly difficult concept. One problem when working with pointers is that students have trouble visualizing the data structures and how their nodes are linked together. So we have included illustrations that show the links and the sequence in which they are created. The binary tree example is a superb capstone for the study of pointers and dynamic data structures. This

example creates a binary tree; enforces duplicate elimination; and introduces recursive pre-order, inorder and postorder tree traversals. Students have a genuine sense of accomplishment when they study and implement this example. They particularly appreciate seeing that the inorder traversal prints the node values in sorted order. The chapter includes a substantial collection of exercises. A highlight of the exercises is the special section “Building Your Own Compiler.” The exercises walk the student through the development of an infix-to-postfix-conversion program and a postfix-expression-evaluation program. We then modify the postfix evaluation algorithm to generate machine-language code. The compiler places this code in a file (using the techniques of Chapter 14). Students then run the machine language produced by their compilers on the software simulators they built in the exercises of Chapter 5! The 67 exercises include a supermarket simulation using queueing, recursively searching a list, recursively printing a list backwards, binary-tree node deletion, level-order traversal of a binary tree, printing trees, writing a portion of an optimizing compiler, writing an interpreter, inserting/deleting anywhere in a linked list, implementing lists and queues without tail pointers, analyzing the performance of binary tree searching and sorting and implementing an indexed list class. After studying Chapter 15, the reader is prepared for the treatment of STL containers, iterators and algorithms in Chapter 20. The STL containers are pre-packaged, templated data structures that most programs will find sufficient for the vast majority of applications they will need to implement. STL is a giant leap forward in achieving the vision of reuse, reuse and reuse.

Chapter 16—Bits, Characters, Strings and Structures—presents a variety of important features. C++’s powerful bit-manipulation capabilities enable programmers to write programs that exercise lower-level hardware capabilities. This helps programs process bit strings, set individual bits on or off and store information more compactly. Such capabilities, often found only in low-level assembly languages, are valued by programmers writing system software such as operating systems and networking software. As you recall, we introduced C-style `char *` string manipulation in Chapter 5 and presented the most popular string-manipulation functions. In Chapter 16, we continue our presentation of characters and C-style `char *` strings. We present the various character-manipulation capabilities of the `<cctype>` library—these include the ability to test a character to see if it is a digit, an alphabetic character, an alphanumeric character, a hexadecimal digit, a lowercase letter, an uppercase letter, etc. We present the remaining string-manipulation functions of the various string-related libraries; as always, every function is presented in the context of a complete, working C++ program. Structures are like records in Pascal and other languages—they aggregate data items of various types. Structures are used in Chapter 14 to form files consisting of records of information. Structures are used in conjunction with pointers and dynamic memory allocation in Chapter 15 to form dynamic data structures such as linked lists, queues, stacks and trees. A feature of the chapter is its high-performance card shuffling and dealing simulation. This is an excellent opportunity for the instructor to emphasize the quality of algorithms. The 38 exercises encourage the student to try out most of the capabilities discussed in the chapter. The feature exercise leads the student through the development of a spell checker program. Chapters 1–5 and 16–18 are mostly the “C legacy” portion of C++. In particular, this chapter presents a deeper treatment of C-like, `char *` strings for the benefit of C++ programmers who are likely to work with C legacy code. Again, Chapter 19 discusses class `string` and discusses manipulating strings as full-fledged objects.

Chapter 17—The Preprocessor—provides detailed discussions of the preprocessor directives. The chapter includes more complete information on the `#include` directive that causes a copy of a specified file to be included in place of the directive before the file is compiled and the `#define` directive that creates symbolic constants and macros. The chapter explains conditional compilation for enabling the programmer to control the execution of preprocessor directives and the compilation of program code. The `#` operator that converts its operand to a string and the `##` operator that concatenates two tokens are discussed. The various predefined preprocessor symbolic constants (`__LINE__`, `__FILE__`, `__DATE__` and `__TIME__`) are presented. Finally, macro `assert` of the header file `<cassert>` is discussed; `assert` is valuable in program testing, debugging, verification and validation. We have used `assert` in many examples, but the reader is urged to begin using exception handling instead, as we discussed in Chapter 13.

Chapter 18—C Legacy Code Topics—presents additional topics including several advanced topics not ordinarily covered in introductory courses. We show how to redirect program input to come from a file, redirect program output to be placed in a file, redirect the output of one program to be the input of another program (piping), append the output of a program to an existing file, develop functions that use variable-length argument lists, pass command-line arguments to function `main` and use them in a program, compile programs whose components are spread across multiple files, register functions with `atexit` to be executed at program termination, terminate program execution with function `exit`, use the `const` and `volatile` type qualifiers, specify the type of a numeric constant using the integer and floating-point suffixes, use the signal-handling library to trap unexpected events, create and use dynamic arrays with `calloc` and `realloc`, use `unions` as a space-saving technique and use linkage specifications when C++ programs are to be linked with legacy C code. As the title suggests, this chapter is intended primarily for C++ programmers who will be working with C legacy code.

Chapter 19—Class `string` and String Stream Processing—The chapter also discusses C++'s capabilities for inputting data from strings in memory and outputting data to strings in memory; these capabilities are often referred to as in-core formatting or string-stream processing. Class `string` is a required component of the Standard Library. Although we placed this material in a chapter near the end of the book, many instructors will want to incorporate the discussion of “strings as full-fledged objects” early in their courses. We preserved the treatment of C-like strings in Chapter 5 and later for several reasons. First, we feel it strengthens the reader's understanding of pointers. Second, we feel that for the next decade, or so, C++ programmers will need to be able to read and modify the enormous amounts of C legacy code that have accumulated over the last quarter of a century and this code processes strings as pointers, as does a large portion of even the C++ code that has been written in industry over the last many years. In Chapter 19 we discuss `string` assignment, concatenation and comparison. We show how to determine various `string` characteristics such as a `string`'s size, capacity and whether or not it is empty. We discuss how to resize a `string`. We consider the various `find` functions that enable us to find a substring in a `string` (searching the `string` either forwards or backwards), and we show how to find either the first occurrence or last occurrence of a character selected from a `string` of characters, and how to find the first occurrence or last occurrence of a character that is not included in a `string`. We show how to replace, erase and insert characters in a `string`. We show how to convert a `string` object to a C-style `char *` string.

Chapter 20—Standard Template Library (STL)—We emphasize here again that this is not an STL book, nor is there any discussion of actual STL features in the first 18 chapters. Chapter 19 does make brief mention of iterators, but states that the real discussion of iterators is in Chapter 20. With the inclusion of Chapter 20, *C++ How to Program: Third Edition* now discusses four programming paradigms: procedural programming, object-based programming, object-oriented programming and generic programming (with the STL). The challenges of teaching object-oriented programming will increase as class libraries and class template libraries grow. We believe that there will be exponential growth in reusable componentry over the next few decades. The early computer science curriculum will need to present the root language, indicate how to craft valuable classes, overview key existing class libraries and show how to reuse these components. Upper-level computer science courses, and, in fact, courses in most any topic for which computers are used (i.e., today that means most any topic, period) will cover their bodies of knowledge and these will include discussion and use of the class libraries that apply to that subject area. Many efforts are underway to support reuse across platforms, so it will not matter what language your classes are written in; you will be able to reuse them from many different languages.

Chapter 21—ANSI/ISO C++ Standard Language Additions—This chapter is a collection of miscellaneous additions to the language. We discuss data type `bool` with data values `false` and `true`—a more natural representation than using non-zero and zero values (although these may still be used). We discuss the four new cast operators: `static_cast`, `const_cast`, `reinterpret_cast` and `dynamic_cast`. These provide a much more robust mechanism for dealing with casts than the style of casts C++ inherited from C. We discuss `namespaces`, a feature particularly crucial for software developers building substantial systems, especially when using a variety of class libraries. Namespaces prevent the kinds of naming collisions that previously hindered such large software efforts. We consider run-time type information (RTTI) which allows existing programs to check the type of an object, something they could not do previously unless the programmer explicitly included a type code (an undesirable programming practice). We discuss the use of operators `typeid` and `dynamic_cast`. We discuss the new operator keywords; these are useful for programmers who do not like cryptic operators, but their primary use is in international markets where certain characters are not normally available on local keyboards. We consider the use of keyword `explicit` that prevents the compiler from invoking conversion constructors when it would be undesirable to do so; `explicit` conversion constructors can only be invoked through constructor syntax, not through implicit conversions. We discuss keyword `mutable`, which allows a member of a `const` object to be changed. Previously this was accomplished by “casting away `const`-ness,” a dangerous practice. We also discuss a few features that are not new, but which we chose not to include in the main portion of the book, because they are relatively obscure, namely pointer-to-member operators `.*` and `->*` and using `virtual` base classes with multiple inheritance.

Appendix A—Operator Precedence Chart—We have reformatted the table to be more useful. Each operator is now on a line by itself with the operator symbol, its name and its associativity.

Appendix B—ASCII Character Set—We resisted the temptation to expand this substantially to include the relatively new international Unicode character set. By the next edition, we expect to discuss Unicode in detail.

Appendix C—Number Systems—discusses the binary, octal, decimal and hexadecimal number systems. It considers how to convert numbers between bases and explains the one’s complement and two’s complement binary representations.

Appendix D—C++ Internet and Web Resources—contains a huge listing of valuable C++ resources such as demos, information about popular compilers (including freebies), books, articles, conferences, job banks, journals, magazines, help, tutorials, FAQs (frequently asked questions), newsgroups, copies of the ANSI/ISO C++ Standard document, Web-based courses, product news and C++ development tools.

Bibliography—lists 125 books and articles—some of historical interest and most quite recent—to encourage the student to do further reading on C++ and OOP.

Index—The book contains a comprehensive index to enable the reader to locate by keyword any term or concept throughout the text.

The C++ Multimedia Cyber Classroom: Third Edition

We have implemented an interactive, CD-ROM-based, software version of *C++ How to Program: Third Edition* called the *C++ Multimedia Cyber Classroom: Third Edition*. It is loaded with features for learning and reference. The *Cyber Classroom* is wrapped with the textbook in a publication called *The Complete C++ Training Course: Third Edition*. If you have already purchased the textbook, you can get a copy of the *C++ Multimedia Cyber Classroom* CD directly from Prentice Hall. Please see the ordering instructions on the last few pages of this book.

There is an introductory presentation in which the authors overview the *Cyber Classroom*’s features. The 268 live-code example C++ programs in the textbook truly “come alive” in the *C++ Multimedia Cyber Classroom*. We have placed executables for all these examples “under the hood” of the *C++ Multimedia Cyber Classroom*, so if you are viewing a program and want to execute it, you simply click the lightning bolt icon and the program runs. You will immediately see—and hear (for the audio-based multimedia programs)—the program’s outputs. If you want to modify a program and see the effects of your changes, simply click the floppy-disk icon that causes the source code to be “lifted off” the CD and “dropped into” one of your own directories so you can edit the text, recompile the program and try out your new version. Click the audio icon and Paul Deitel will talk about the program and “walk you through” the code. (You will not hear Harvey Deitel’s voice in these audios—our friends at Prentice Hall like Paul’s voice better!)

C++ How to Program: Third Edition contains thousands of exercises that are divided into groups and have varying levels of difficulty. Approximately half of the solutions are provided on the *C++ Multimedia Cyber Classroom* (the remaining exercise solutions are reserved for instructors who wish to assign these exercises as homework). For the *Cyber Classroom*, we carefully selected at least one exercise solution from each group. This allows a student looking at the solution for one exercise in a group to apply the techniques shown in that solution to other exercises.

The *C++ Multimedia Cyber Classroom* provides various navigational aids including extensive hyperlinking. The *C++ Multimedia Cyber Classroom* remembers in a “history list” recent sections you have visited and allows you to move forward or backward in that history list. The thousands of index entries are hyperlinked to their text occurrences. You

can key in a term and the *C++ Multimedia Cyber Classroom* will locate its occurrences throughout the text. The *Table of Contents* entries are “hot,” so clicking a chapter or section name immediately takes you to that chapter or section. You can insert “bookmarks” at places to which you may want to return.

This third edition Cyber Classroom has been completely redesigned using a Web-browser-based interface. The Cyber Classroom comes in two forms—a CD-ROM version for Microsoft® Windows® platforms and a Web-based version. The Web-based version is ideal for students who prefer the convenience of Internet delivery or who want to run the Cyber Classroom on non-Windows platforms.

Students and professional users of our *Cyber Classrooms* tell us they like the interactivity and that the *Cyber Classroom* is an effective reference because of the extensive hyperlinking and other navigational features. We received an email from a person who said that he lives “in the boonies” and cannot take a live course at a college, so the *Cyber Classroom* was a nice solution to his educational needs.

Professors have sent us emails indicating their students enjoy using the *Cyber Classroom*, spend more time on the course and master more of the material than in textbook-only courses. Also, the *Cyber Classroom* helps shrink lines outside professors’ offices during office hours. Prentice Hall is now publishing Cyber Classrooms for most of our books—we will soon publish our books in Web-based training (WBT) format.

Acknowledgments

One of the great pleasures of writing a textbook is acknowledging the efforts of many people whose names may not appear on the cover, but whose hard work, cooperation, friendship and understanding were crucial to the production of the book.

Many other people at Deitel & Associates, Inc. devoted long hours to this project.

- Tem Nieto, a graduate of the Massachusetts Institute of Technology, is one of our full-time colleagues at Deitel & Associates, Inc. and was recently promoted to Director of Product Development. Tem teaches C++, C and Java seminars and works with us on textbook writing, course development and multimedia authoring efforts. Tem co-authored Chapter 19, Chapter 21 and the Special Section entitled “Building Your Own Compiler” in Chapter 15. He also contributed to the Instructor’s Manual and the *C++ Multimedia Cyber Classroom: Third Edition*.
- Barbara Deitel managed the preparation of the manuscript and coordinated with Prentice Hall all the efforts related to production of the book. Barbara’s efforts are by far the most painstaking of what we do to develop books. She has infinite patience. She handled the endless details involved in publishing a 1200-page, four-color book; a 550-page instructor’s manual and the 650 megabyte CD *C++ Multimedia Cyber Classroom*. She spent long hours researching the quotations at the beginning of each chapter. She did all this in parallel with handling her extensive financial and administrative responsibilities at Deitel & Associates, Inc.
- Abbey Deitel, a graduate of Carnegie Mellon University’s industrial management program, and now President and Director of Worldwide Marketing at Deitel & Associates, Inc., wrote Appendix D and suggested the title for the book. We asked Abbey to surf the World Wide Web and track down the best C++ sites. She used every major Web search engine and collected this information for you in Appen-

dix D. For each resource and demo, Abbey has provided a brief explanation. She rejected hundreds of sites and has listed for you the best she could find. Abbey will be keeping this resources and demos listing on our Web site www.deitel.com. Please send URLs for your favorite sites to her by email at deitel@deitel.com and she will post links to these on our site.

Deitel & Associates, Inc. student interns who worked on this book include:

- Ben Wiedermann—a computer science major at Boston University—was the lead developer, programmer and writer working with Dr. Harvey M. Deitel on the UML case study. We wish to acknowledge Ben’s extraordinary commitment and contributions to this project.
- Sean Santry—a computer science and philosophy graduate of Boston College—worked on the coding and code walkthroughs of the UML Case Study. Sean has joined Deitel & Associates, Inc. full time and is working as a lead developer with Paul Deitel on our forthcoming book, *Advanced Java How to Program*.
- Blake Perdue—a computer science major at Vanderbilt University—helped develop the UML Case Study.
- Kalid Azad—a computer science major at Princeton University—worked extensively on the book’s ancillaries including the PowerPoint® Instructor Lecture Notes and the test bank.
- Aftab Bukhari—a computer science major at Boston University—performed extensive program testing and verification and worked on the book’s ancillaries including the PowerPoint Instructor Lecture Notes and the Instructor’s Manual.
- Jason Rosenfeld—a computer science major at Northwestern University—worked on the book’s ancillaries including the Instructor’s Manual.
- Melissa Jordan—a graphic design major at Boston University—colored the art for the entire book and created several original illustrations.
- Rudolf Faust—a freshman at Stanford University—helped create the test bank.

We are fortunate to have been able to work on this project with a talented and dedicated team of publishing professionals at Prentice Hall. This book happened because of the encouragement, enthusiasm and persistence of our computer science editor, Petra Recter, and her boss—the best friend we have had in 25 years of publishing—Marcia Horton, Editor-in-Chief of Prentice-Hall’s Engineering and Computer Science Division. Camille Trentacoste did a marvelous job as production manager. Sarah Burrows did a marvelous job with her work on both the review process and the book supplements.

The *C++ Multimedia Cyber Classroom: Third Edition* was developed in parallel with *C++ How to Program: Third Edition*. We sincerely appreciate the “new media” insight, savvy and technical expertise of our editor Mark Taub and his colleague Karen McLean. Mark and Karen did a remarkable job bringing the *C++ Multimedia Cyber Classroom: Third Edition*, to publication under a tight schedule. They are surely among the world’s leaders in new-media publishing.

We owe special thanks to the creativity of Tamara Newnam Cavallo (smart-art@earthlink.net) who did the art work for our programming tips icons and the cover. She created the delightful creature who shares with you the book’s programming tips. Please help us name this endearing little bug. Some early suggestions: D. Bug,

InterGnat, Ms. Kito, DeetleBug (an unfortunate moniker that was attached to the old guy in high school) and Feature (“It’s not a bug, it’s a feature”).

We wish to acknowledge the efforts of our *Third Edition* reviewers and to give a special note of thanks to Crissy Statuto of Prentice Hall who managed this extraordinary review effort.

Reviewers of C++ Material

- Tamer Nassif (Motorola)
- Christophe Dinechin (Hewlett Packard)
- Thomas Kiesler (Montgomery College)
- Mary Astone (Troy State University)
- Simon North (Synopsis)
- Harold Howe (Inprise)
- William Hasserman (University of Wisconsin)
- Phillip Wasserman (Chabot College)
- Richard Albright (University of Delaware)
- Mahe Velauthapilla (Georgetown University)
- Chris Uzdavinis (Automated Trading Desk)
- Stephen Clamage (Chairman of ANSI C++ Standards Committee)
- Ram Chopra (Akili Systems; University of Houston)
- Wolfgang Pelz (University of Akron)

Reviewers of the UML Case Study

- Spencer Roberts (Titus Corporation)
- Don Kostuch (You Can C Clearly Now)
- Kendall Scott (Independent consultant; UML author)
- Grant Larsen (Blueprint Technologies)
- Brian Cook (Technical Resource Connection; OMG)
- Michael Chonoles (Chief of Methodology, Lockheed Martin Adv. Concepts; OMG)
- Stephen Tockey (Construx Software; OMG)
- Cameron Skinner (Advanced Software Technologies; OMG)
- Rick Cassidy (Advanced Concepts Center)
- Mark Contois - NetBeans
- David Papurt (Independent consultant; C++ lecturer and author)
- Chris Norton (AD2IT; Independent consultant)

We wish to acknowledge again the efforts of our previous edition reviewers (some first edition, some second edition and some both):

- Richard Albright (University of Delaware)
- Ken Arnold (Sun Microsystems)
- Ian Baker (Microsoft)
- Pete Becker (Member of ANSI/ISO C++ Committee; Dinkumware, LTD.)
- Timothy D. Born (Delta C-Fax)

- John Carson (George Washington University)
- Steve Clamage (Chairman of ANSI/ISO C++ Standards Committee; Sunsoft)
- Marian Corcoran (Member ANSI/ISO C++ Standards Committee)
- Edgar Crisostomo (Siemens/Rolm)
- David Finkel (Worcester Polytechnic Institute)
- Rex Jaeschke (Chairman, ANSI/ISO C Committee)
- Frank Kelbe (Naval Postgraduate School)
- Chris Kelsey (Kelsey Associates)
- Don Kostuch (You Can C Clearly Now)
- Meng Lee (Co-creator of STL; Hewlett-Packard)
- Barbara Moo (AT&T Bell Labs)
- David Papurt (Consultant)
- Wolfgang Pelz (University of Akron)
- Jandelyn Plane (University of Maryland College Park)
- Paul Power (Borland)
- Kenneth Reek (Rochester Institute of Technology)
- Larry Rosler (Hewlett-Packard)
- Robin Rowe (Halycon/Naval Postgraduate School)
- Brett Schuchert (ObjectSpace; Co-Authored *STL Primer*)
- Alexander Stepanov (Co-creator of STL; Silicon Graphics)
- William Tepfenhart (AT&T; Author *UML and C++: A Practical Guide to Object-Oriented Development*)
- David Vandevoorde (Member of the ANSI/ISO C++ Committee; Hewlett-Packard)
- Terry Wagner (University of Texas)

Under tight deadlines, they scrutinized every aspect of the text and made countless suggestions for improving the accuracy and completeness of the presentation.

We would sincerely appreciate your comments, criticisms, corrections and suggestions for improving the text. Please address all correspondence to:

`deitel@deitel.com`

We will respond immediately. Well, that is it for now. Welcome to the exciting world of C++, object-oriented programming, the UML and generic programming with the STL. We hope you enjoy this look at contemporary computer programming. Good luck!

Dr. Harvey M. Deitel
Paul J. Deitel

About the Authors

Dr. Harvey M. Deitel, CEO of Deitel & Associates, Inc., has 39 years experience in the computing field including extensive industry and academic experience. He is one of the world's leading computer science instructors and seminar presenters. Dr. Deitel earned

B.S. and M.S. degrees from the Massachusetts Institute of Technology and a Ph.D. from Boston University. He worked on the pioneering virtual memory operating systems projects at IBM and MIT that developed techniques widely implemented today in systems like UNIX[®], Windows NT,[™] OS/2 and Linux. He has 20 years of college teaching experience including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc. with Paul J. Deitel. He is author or co-author of several dozen books and multimedia packages and is currently writing five more. With translations published in Japanese, Russian, Spanish, Elementary Chinese, Advanced Chinese, Korean, French, Portuguese, Polish and Italian the Deitels' texts have earned international recognition.

Paul J. Deitel, Executive Vice President of Deitel & Associates, Inc., is a graduate of the Massachusetts Institute of Technology's Sloan School of Management where he studied Information Technology. Through Deitel & Associates, Inc. he has delivered Java, C, C++, Internet and World Wide Web courses for industry clients including Compaq, Sun Microsystems, White Sands Missile Range, Rogue Wave Software, Computervision, Stratus, Fidelity, Cambridge Technology Partners, Open Environment Corporation, One Wave, Hyperion Software, Lucent Technologies, Adra Systems, Entergy, CableData Systems, NASA at the Kennedy Space Center, the National Severe Storm Center, IBM and many other organizations. He has lectured on C++ and Java for the Boston Chapter of the Association for Computing Machinery, and has taught satellite-based Java courses through a cooperative venture of Deitel & Associates, Inc., Prentice Hall and the Technology Education Network.

The Deitels are co-authors of the best-selling introductory college computer-science programming language textbooks, *C How to Program: Third Edition*, *Java How to Program: Third Edition*, *Visual Basic 6 How to Program (co-authored with Tem R. Nieto)* and *Internet and World Wide Web How to Program (co-authored with Tem R. Nieto)*. The Deitels are also co-authors of the *C++ Multimedia Cyber Classroom: Third Edition* (the first edition of this was Prentice Hall's first multimedia-based textbook), the *Java 2 Multimedia Cyber Classroom: Third Edition*, the *Visual Basic 6 Multimedia Cyber Classroom* and the *Internet and World Wide Web Programming Multimedia Cyber Classroom*. The Deitels are also co-authors of *The Complete C++ Training Course: Third Edition*, *The Complete Visual Basic 6 Training Course*, *The Complete Java 2 Training Course: Third Edition* and *The Complete Internet and World Wide Web Programming Training Course*—these products each contain the corresponding *How to Program Series* textbook and the corresponding *Multimedia Cyber Classroom*.

About Deitel & Associates, Inc.

Deitel & Associates, Inc. is a rapidly growing, internationally recognized corporate training and publishing organization specializing in programming languages, Internet, World Wide Web and object technology education. The company provides courses on C++, Java, C, Visual Basic, Internet and World Wide Web programming and object-technology. The principals of Deitel & Associates, Inc. are Dr. Harvey M. Deitel and Paul J. Deitel. The company's clients include some of the world's largest computer companies, government agencies and business organizations. Through its publishing partnership with Prentice Hall, Deitel & Associates, Inc. publishes leading-edge programming textbooks and professional

books, interactive CD-ROM-based multimedia *Cyber Classrooms* and Web-based training courses. Deitel & Associates, Inc. and the authors can be reached via email at

`deitel@deitel.com`

To learn more about Deitel & Associates, Inc., its publications and its on-site course curriculum, visit:

`www.deitel.com`

To learn more about Deitel/Prentice Hall publications, visit:

`www.prenhall.com/deitel`

For a current list of Deitel/Prentice Hall publications including textbooks and *Cyber Classrooms*, *Complete Training Courses* and Web-Based Training products, and for complete worldwide ordering information, please see the last few pages of this book.