Preface

"The chief merit of language is clearness …"
—Galen

Welcome to C++ and *C++ How to Program, Sixth Edition*! At Deitel & Associates, we write programming language textbooks and professional books for Prentice Hall, deliver corporate training worldwide and develop Web 2.0 Internet businesses. This book reflects significant changes to the C++ language and to the preferred ways of teaching and learning programming. All the chapters have been significantly tuned. The *Tour of the Book* section of the Preface gives instructors, students and professionals a sense of *C++ How to Program, 6/e*'s coverage of C++ and object-oriented programming.

New and Updated Features

Here's a list of updates we've made to the fifth and sixth editions of *C++ How to Program*:

- **Game Programming.** We've added a new chapter on game programming. The computer-game industry's revenues are already greater than those of the first-run movie business, creating lots of opportunities for students interested in game-programming careers. Chapter 23, Game Programming with Ogre, introduces game programming and graphics with the open source Ogre 3D graphics engine. We discuss basic issues involved in game programming. Then we show how to use Ogre to create a simple game featuring a play mechanic similar to the classic video game Pong®, originally developed by Atari in 1972. We demonstrate how to create a scene with colored 3D graphics, smoothly animate moving objects, use timers to control animation speed, detect collisions between objects, add sound, accept keyboard input and display text output.
- **Future of C++.** We've added Chapter 24, which considers the future of C++—we introduce the Boost C++ Libraries, Technical Report 1 (TR1) and C++0x. The free Boost open source libraries are created by members of the C++ community. Technical Report 1 describes the proposed changes to the C++ Standard Library, many of which are based on current Boost libraries. The C++ Standards Committee is revising the C++ Standard. The main goals for the new standard are to make C++ easier to learn, improve library building capabilities, and increase compatibility with the C programming language. The last standard was published in 1998. Work on the new standard, currently referred to as C++0x, began in 2003. The new standard is likely to be released in 2009. It will include changes to the core language and, most likely, many of the libraries in TR1. We overview the TR1 libraries and provide code examples for the "regular expression" and "smart pointer" libraries. Regular expressions are used to match specific character patterns in text. They can be used to validate data to ensure that it is in

a particular format, to replace parts of one string with another, or to split a string. Many common bugs in C and C++ code are related to pointers, a powerful programming capability you'll study in Chapter 8, Pointers and Pointer-Based Strings. Smart pointers help you avoid errors by providing additional functionality to standard pointers.

- **Major Content Revisions.** All the chapters have been significantly updated and upgraded. We tuned the writing for clarity and precision. We also adjusted our use of C++ terminology in accordance with the ISO/IEC C++ standard document that defines the language.
- **Early Classes and Objects Approach.** Students are introduced to the basic concepts and terminology of object technology in Chapter 1 and begin developing customized, reusable classes and objects in Chapter 3. This book presents object-oriented programming, where appropriate, from the start and throughout the text. The early discussion of objects and classes gets students "thinking about objects" immediately and mastering these concepts more completely. Object-oriented programming is not trivial by any means, but it's fun to write object-oriented programs, and students can see immediate results.
- **Integrated Case Studies.** We provide several case studies spanning multiple sections and chapters that often build on a class introduced earlier in the book to demonstrate new programming concepts later in the book. These case studies include the development of the `GradeBook` class in Chapters 3–7, the `Time` class in several sections of Chapters 9–10, the `Employee` class in Chapters 12–13, and the optional OOD/UML ATM case study in Chapters 1–7, 9, 13 and Appendix G.
- **Integrated `GradeBook` Case Study.** The `GradeBook` case study reinforces our early classes presentation. It uses classes and objects in Chapters 3–7 to incrementally build a `GradeBook` class that represents an instructor's grade book and performs various calculations based on a set of student grades, such as calculating the average grade, finding the maximum and minimum, and printing a bar chart.
- **Unified Modeling Language™ 2 (UML 2).** The Unified Modeling Language (UML) has become the preferred graphical modeling language for designers of object-oriented systems. All the UML diagrams in the book comply with the UML 2 specification. We use UML class diagrams to visually represent classes and their inheritance relationships, and we use UML activity diagrams to demonstrate the flow of control in each of C++'s control statements. We make especially heavy use of the UML in the optional OOD/UML ATM case study
- **Optional OOD/UML ATM Case Study.** The optional OOD/UML automated teller machine (ATM) case study in the Software Engineering Case Study sections of Chapters 1–7, 9 and 13 is appropriate for first and second programming courses. The nine case study sections present a carefully paced introduction to object-oriented design using the UML. We introduce a concise, simplified subset of the UML 2, then guide you through a first design experience intended for the novice object-oriented designer/programmer. Our goal in this case study is to help students develop an object-oriented design to complement the object-oriented programming concepts they begin learning in Chapter 1 and implementing in

Chapter 3. The case study was reviewed by a distinguished team of OOD/UML academic and industry professionals. The case study is not an exercise; rather, it is a fully developed end-to-end learning experience that concludes with a detailed walkthrough of the complete 877-line C++ code implementation. We take a detailed tour of the nine sections of this case study later in the Preface.

- **Compilation and Linking Process for Multiple-Source-File Programs.** Chapter 3 includes a detailed diagram and discussion of the compilation and linking process that produces an executable application.
- **Function Call Stack Explanation.** In Chapter 6, we provide a detailed discussion (with illustrations) of the function call stack and activation records to explain how C++ is able to keep track of which function is currently executing, how automatic variables of functions are maintained in memory and how a function knows where to return after it completes execution.
- **C++ Standard Library `string` and `vector` Classes.** The `string` and `vector` classes are used to make earlier examples more object-oriented.
- **Class `string`.** We use class `string` instead of C-like pointer-based `char *` strings for most string manipulations throughout the book. We continue to include discussions of `char *` strings in Chapters 8, 10, 11 and 21 to give students practice with pointer manipulations, to illustrate dynamic memory allocation with `new` and `delete`, to build our own `String` class, and to prepare students for working with `char *` strings in C and C++ legacy code.
- **Class Template `vector`.** We use class template `vector` instead of C-like pointer-based array manipulations throughout the book. However, we begin by discussing C-like pointer-based arrays in Chapter 7 to prepare students for working with C and C++ legacy code and to use as a basis for building our own customized `Array` class in Chapter 11, Operator Overloading; String and Array Objects.
- **Tuned Treatment of Inheritance and Polymorphism.** Chapters 12–13 have been carefully tuned using an `Employee` class hierarchy to make the treatment of inheritance and polymorphism clearer and more accessible for students who are new to OOP.
- **Discussion and Illustration of How Polymorphism Works "Under the Hood."** Chapter 13 contains a detailed diagram and explanation of how C++ can implement polymorphism, `virtual` functions and dynamic binding internally. This gives students a solid understanding of how these capabilities really work. More importantly, it helps students appreciate the overhead of polymorphism—in terms of additional memory consumption and processor time. This helps students determine when to use polymorphism and when to avoid it.
- **Standard Template Library (STL).** This might be one of the most important topics in the book in terms of your appreciation of software reuse. The STL defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures. Chapter 22 introduces the STL and discusses its three key components—containers, iterators and algorithms. We show that using STL components provides tremendous expressive power, often reducing many lines of code to a single statement.

- **ISO/IEC C++ Standard Compliance.** We have audited our presentation against the most recent ISO/IEC C++ standard document for completeness and accuracy. [*Note:* If you need additional technical details on C++, you may want to read the C++ standard document. A PDF copy of the C++ standard (document number INCITS/ISO/IEC 14882-2003) can be purchased at `webstore.ansi.org/ansidocstore/default.asp`.]
- **Debugger Appendices.** We include two Using the Debugger appendices—Appendix I, Using the Visual Studio Debugger, and Appendix J, Using the GNU C++ Debugger.
- **Code Testing on Multiple Platforms.** We tested the code examples on various popular C++ platforms. For the most part, all of the book's examples port easily to all popular standard-compliant compilers.
- **Errors and Warnings Shown for Multiple Platforms.** For programs that intentionally contain errors to illustrate a key concept, we show the error messages that result on several popular platforms.

All of this has been carefully reviewed by distinguished academics and industry developers who worked with us on *C++ How to Program, 5/e* and *C++ How to Program, 6/e*.

We believe that this book and its support materials will provide students and professionals with an informative, interesting, challenging and entertaining C++ educational experience. The book includes a comprehensive suite of ancillary materials that help instructors maximize their students' learning experience.

As you read this book, if you have questions, send an e-mail to `deitel@deitel.com`; we'll respond promptly. For updates on this book and the status of all supporting C++ software, and for the latest news on all Deitel publications and services, visit `www.deitel.com`. Sign up at `www.deitel.com/newsletter/subscribe.html` for the free *Deitel® Buzz Online* e-mail newsletter and check out our growing list of C++ and related Resource Centers at `www.deitel.com/ResourceCenters.html`. Each week we announce our latest Resource Centers in the newsletter. Please let us know of other Resource Centers you'd like to see.

Teaching Approach

C++ How to Program, 6/e contains a rich collection of examples. The book concentrates on the principles of good software engineering and stresses program clarity. We teach by example. We are educators who teach leading-edge topics in industry classrooms worldwide. Dr. Harvey M. Deitel has 20 years of college teaching experience and 18 years of industry teaching experience. Paul Deitel has 16 years of industry teaching experience. The Deitels have taught courses at all levels to government, industry, military and academic clients of Deitel & Associates.

Live-Code Approach. *C++ How to Program, 6/e* is loaded with "live-code" examples—by this we mean that each new concept is presented in the context of a complete working C++ application that is immediately followed by one or more actual executions showing the program's inputs and outputs. This style exemplifies the way we teach and write about programming; we call this the "live-code approach."

Syntax Coloring. We syntax color all the C++ code, similar to the way most C++ integrated-development environments and code editors syntax color code. This improves code readability—an important goal, given that this book contains about 20,000 lines of code in complete, working C++ programs. Our syntax-coloring conventions are as follows:

```
comments appear in green
keywords appear in dark blue
errors appear in red
constants and literal values appear in light blue
all other code appears in black
```

Code Highlighting. We place gray rectangles around the key code segments in each program.

Using Fonts and Colors for Emphasis. We place the key terms and the index's page reference for each defining occurrence in **bold blue** text for easier reference. We emphasize on-screen components in the **bold Helvetica** font (e.g., the **File** menu) and emphasize C++ program text in the `Lucida` font (e.g., `int x = 5`).

Web Access. All of the source-code examples for *C++ How to Program, 6/e* are available for download from:

```
www.deitel.com/books/cpphtp6/
```

Site registration is quick and easy. Download all the examples, then run each program as you read the corresponding text discussions. Making changes to the examples and seeing the effects of those changes is a great way to enhance your C++ learning experience.

Objectives. Each chapter begins with a statement of objectives. This lets you know what to expect and gives you an opportunity, after reading the chapter, to determine if you have met the objectives.

Quotations. The learning objectives are followed by quotations. Some are humorous; some are philosophical; others offer interesting insights. We hope that you enjoy relating the quotations to the chapter material.

Outline. The chapter outline helps you approach the material in a top-down fashion, so you can anticipate what is to come and set a comfortable and effective learning pace.

Illustrations/Figures. Abundant charts, tables, line drawings, programs and program output are included. We model the flow of control in control statements with UML activity diagrams. UML class diagrams model the fields, constructors and methods of classes. We make extensive use of six major UML diagram types in the optional OOD/UML 2 ATM case study.

Programming Tips. We include programming tips to help you focus on important aspects of program development. These tips and practices represent the best we have gleaned from a combined six decades of programming and teaching experience. One of our students—a mathematics major—told us that she feels this approach is like the highlighting of axioms, theorems and corollaries in mathematics books; it provides a basis on which to build good software.

Good Programming Practices

Good Programming Practices *call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.*

Common Programming Errors

Students tend to make certain kinds of errors frequently. Pointing out these Common Programming Errors *reduces the likelihood that you'll make the same mistakes.*

Error-Prevention Tips

These tips contain suggestions for exposing bugs and removing them from your programs; many describe aspects of C++ that prevent bugs from getting into programs in the first place.

Performance Tips

Students like to "turbo charge" their programs. These tips highlight opportunities for making your programs run faster or minimizing the amount of memory that they occupy.

Portability Tips

We include Portability Tips *to help you write code that will run on a variety of platforms and to explain how C++ achieves its high degree of portability.*

Software Engineering Observations

The Software Engineering Observations *highlight architectural and design issues that affect the construction of software systems, especially large-scale systems.*

Wrap-Up Section. Each of the chapters ends with a brief "wrap-up" section that recaps the chapter content and transitions to the next chapter.

Summary Bullets. Each chapter ends with additional pedagogical features. We present a thorough, bullet-list-style summary of the chapter, section by section.

Terminology. We include an alphabetized list of the important terms defined in each chapter. Each term also appears in the index, with its defining occurrence highlighted with a **bold, blue** page number.

Self-Review Exercises and Answers. Extensive self-review exercises and answers are included for self-study.

Exercises. Each chapter concludes with a substantial set of exercises including simple recall of important terminology and concepts; identifying the errors in code samples; writing individual C++ statements; writing small portions of 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 students and to vary course assignments each semester. Instructors can use these exercises to form homework assignments, short quizzes, major examinations and term projects. See our Programming Projects Resource Center (`www.deitel.com/ProgrammingProjects/`) for many additional exercise and project possibilities.

[***NOTE:*** **Please do not write to us requesting access to the Prentice Hall Instructor's Resource Center. Access is limited strictly to college instructors teaching from the book. Instructors may obtain access only through their Prentice Hall representatives.**]

Thousands of Index Entries. We have included an extensive index which is especially useful when you use the book as a reference.

"Double Indexing" of C++ Live-Code Examples. For every source-code program in the book, we index the figure caption both alphabetically and as a subindex item under "Examples." This makes it easier to find examples using particular features.

Tour of the Book

You'll now take a tour of the C++ capabilities you'll study in *C++ How to Program, 6/e*. Figure 1 illustrates the dependencies among the chapters. We recommend studying the topics in the order indicated by the arrows, though other orders are possible. This book is widely used in all levels of C++ programming courses. Search the Web for "syllabus," "C++" and "Deitel" to find syllabi used with recent editions of this book.

Chapter 1, Introduction to Computers, the Internet and the World Wide Web, discusses what computers are, how they work and how they are programmed. The chapter gives a brief history of the development of programming languages from machine languages to assembly languages and high-level languages. The origin of the C++ programming language is discussed. The chapter includes an introduction to a typical C++ programming environment. We walk readers through a "test drive" of a typical C++ application on Windows and Linux platforms. This chapter also introduces basic object technology concepts and terminology and the Unified Modeling Language.

Chapter 2, Introduction to C++ Programming, provides a lightweight introduction to programming applications in the C++ programming language. The chapter introduces nonprogrammers to basic programming concepts and constructs. The programs in this chapter illustrate how to display data on the screen and how to obtain data from the user at the keyboard. Chapter 2 ends with detailed treatments of decision making and arithmetic operations.

Chapter 3, Introduction to Classes and Objects, provides a friendly early introduction to classes and objects. Chapter 3 gets students working with object orientation comfortably from the start. It was developed with the guidance of a distinguished team of industry and academic reviewers. We introduce classes, objects, member functions, constructors and data members using a series of simple real-world examples. We develop a well-engineered framework for organizing object-oriented programs in C++. First, we motivate the notion of classes with a simple example. Then we present a carefully paced sequence of seven complete working programs to demonstrate creating and using your own classes. These examples begin our **integrated case study on developing a grade-book class** that instructors can use to maintain student test scores. This case study is enhanced over the next several chapters, culminating with the version presented in Chapter 7, Arrays and Vectors. The `GradeBook` class case study describes how to define a class and how to use it to create an object. The case study discusses how to declare and define member functions to implement the class's behaviors, how to declare data members to implement the class's attributes and how to call an object's member functions to make them perform their tasks. We introduce C++ Standard Library class `string` and create `string` objects to store the name of the course that a `GradeBook` object represents. Chapter 3 explains the differences between data members of a class and local variables of a function, and how to use a constructor to ensure that an object's data is initialized when the object is created. We

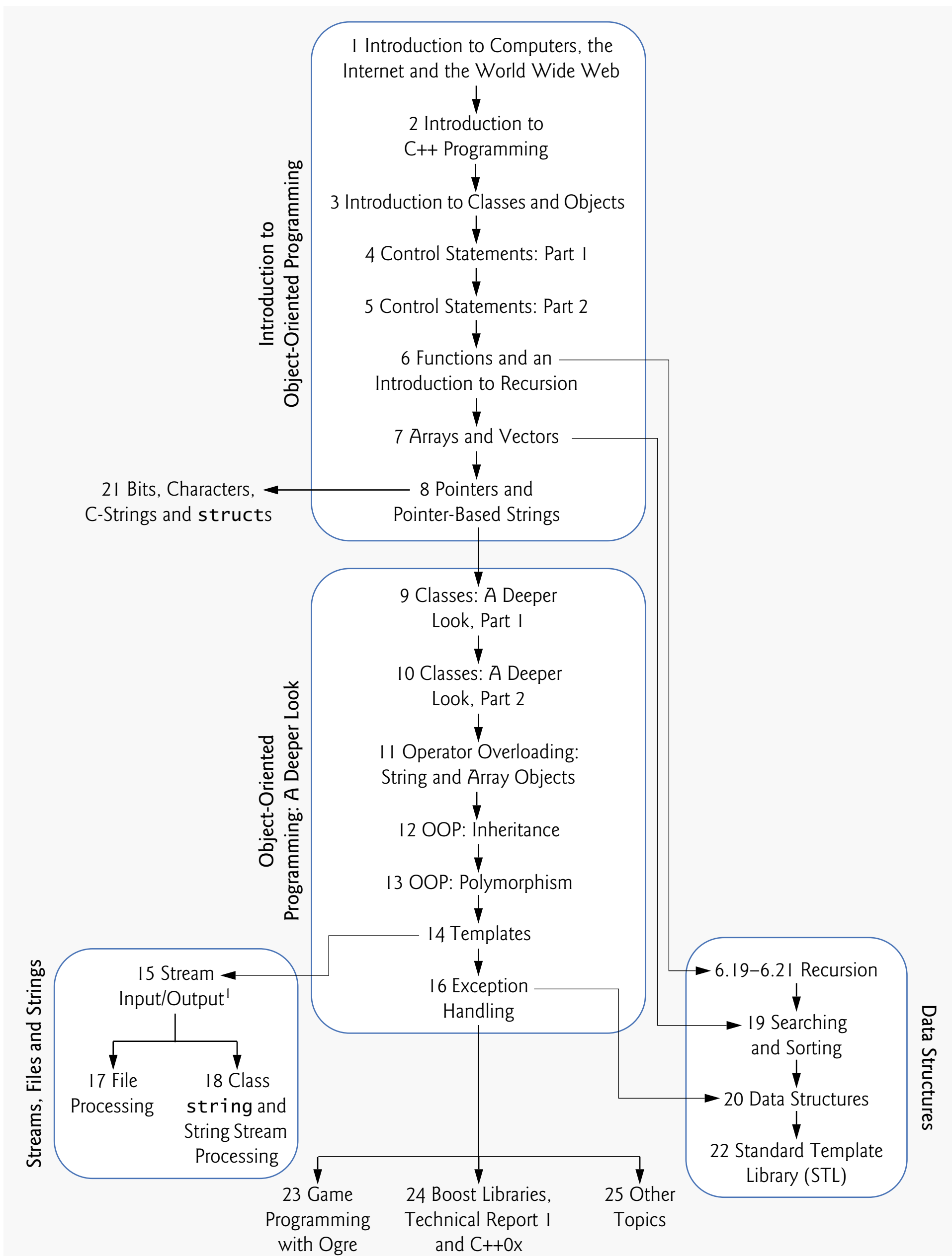

Fig. 1 | *C++ How to Program, 6/e* chapter dependency chart.

show how to promote software reusability by separating a class definition from the client code (e.g., function `main`) that uses the class. We also introduce another fundamental prin-

ciple of good software engineering—separating interface from implementation. The chapter includes a detailed diagram and discussion explaining the compilation and linking process that produces an executable application.

Chapter 4, Control Statements: Part 1, focuses on the program-development process involved in creating useful classes. The chapter discusses how to take a problem statement and develop a working C++ program from it, including performing intermediate steps in pseudocode. The chapter introduces some simple control statements for decision making (`if` and `if...else`) and repetition (`while`). We examine counter-controlled and sentinel-controlled repetition using the **`GradeBook` class** from Chapter 3, and introduce C++'s increment, decrement and assignment operators. The chapter includes **two enhanced versions of the `GradeBook` class**, each based on Chapter 3's final version. These versions each include a member function that uses control statements to calculate the average of a set of student grades. In the first version, the member function uses counter-controlled repetition to input 10 student grades from the user, then determines the average grade. In the second version, the member function uses sentinel-controlled repetition to input an arbitrary number of grades from the user, then calculates the average of the grades that were entered. The chapter uses simple UML activity diagrams to show the flow of control through each of the control statements.

Chapter 5, Control Statements: Part 2, continues the discussion of C++ control statements with examples of the `for` repetition statement, the `do...while` repetition statement, the `switch` selection statement, the `break` statement and the `continue` statement. We create an **enhanced version of class `GradeBook`** that uses a `switch` statement to count the number of A, B, C, D and F grades entered by the user. This version uses sentinel-controlled repetition to input the grades. While reading the grades from the user, a member function modifies data members that keep track of the count of grades in each letter grade category. Another member function of the class then uses these data members to display a summary report based on the grades entered. The chapter includes a discussion of logical operators.

Chapter 6, Functions and an Introduction to Recursion, takes a deeper look inside objects and their member functions. We discuss C++ Standard Library functions and examine more closely how students can build their own functions. The techniques presented in Chapter 6 are essential to the production of properly organized 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. The chapter's first example continues the **`GradeBook` class case study** with an example of a function with multiple parameters. Students will enjoy the chapter's treatment of random numbers and simulation, and the discussion of the dice game of craps, which makes elegant use of control statements. 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. We also present C++'s call-by-value and call-by-reference capabilities. The header files table introduces many of the header files that you'll use throughout the book. In this new edition, we provide a detailed discussion (with illustrations) of the function call stack and activation records to explain how C++ is able to keep track of which function is currently executing, how automatic variables of functions are maintained in

memory and how a function knows where to return after it completes execution. The chapter then offers a solid introduction to recursion and includes a table summarizing the 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 exercises at the end of the chapter includes several classic recursion problems, including the Towers of Hanoi.

Chapter 7, Arrays and Vectors, explains how to process lists and tables of values. We discuss the structuring of data in arrays of data items of the same type and demonstrate how arrays facilitate the tasks performed by objects. The early parts of this chapter use C-style, pointer-based arrays, which, as you'll see in Chapter 8, can be treated as pointers to the array contents in memory. We then present arrays as full-fledged objects, introducing the C++ Standard Library `vector` class template—a robust array data structure. The chapter presents numerous examples of both one-dimensional arrays and two-dimensional arrays. Examples in the chapter investigate various common array manipulations, printing bar charts, sorting data and passing arrays to functions. The chapter includes the **final two `GradeBook` case study sections**, in which we use arrays to store student grades for the duration of a program's execution. Previous versions of the class process a set of grades entered by the user, but do not maintain the individual grade values in data members of the class. In this chapter, we use arrays to enable an object of the `GradeBook` class to maintain a set of grades in memory, thus eliminating the need to repeatedly input the same set of grades. The first version of the class stores the grades in a one-dimensional array and can produce a report containing the average of the grades, the minimum and maximum grades and a bar chart representing the grade distribution. The second version (i.e., the final version in the case study) uses a two-dimensional array to store the grades of a number of students on multiple exams in a semester. This version can calculate each student's semester average, as well as the minimum and maximum grades across all grades received for the semester. The class also produces a bar chart displaying the overall grade distribution for the semester. Another key feature of this chapter is the discussion of elementary sorting and searching techniques. The end-of-chapter exercises include a variety of interesting and challenging problems, such as improved sorting techniques, the design of a simple 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 widely employed in the field of artificial intelligence. The exercises conclude with many recursion problems including selection sort, palindromes, linear search, the Eight Queens, printing an array, printing a string backwards and finding the minimum value in an array.

Chapter 8, Pointers and Pointer-Based Strings, presents one of the most powerful features of the C++ language—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. We demonstrate how to use `const` with pointers to enforce the principle of least privilege to build more robust software. We discuss using the `sizeof` operator to determine the size of a data type or data items in bytes during program compilation. There is an intimate relationship between pointers, arrays and C-style strings in C++, so we introduce basic C-style string-manipulation concepts and discuss some of the most popular C-style string-handling functions, such as `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 (return the length of a string). In this new edition, we frequently use string objects (introduced in Chapter 3, Introduction to Classes and Objects) in place of C-style, char * pointer-based strings. However, we include char * strings in Chapter 8 to help you master pointers and prepare for the professional world in which you'll see a great deal of C legacy code that has been implemented over the last three decades. Thus, you'll become familiar with the two most prevalent methods of creating and manipulating strings in C++. 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 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 pointers. The chapter is loaded with challenging exercises. The 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** also is included. This section explains machine-language programming and proceeds with a project involving the design and implementation of a computer simulator that leads the student to write and run machine-language programs. This unique feature of the text will be especially useful to readers who want to understand how computers really work. Our students enjoy this project and often implement substantial enhancements, many of which are suggested in the exercises. 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.

Chapter 9, Classes: A Deeper Look, Part 1, continues our discussion of object-oriented programming. This chapter uses a rich Time class case study to illustrate 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. Students learn the order in which constructors and destructors are called during the lifetime of an object. A modification of the Time case study demonstrates the problems that can occur when a member function returns a reference to a private data member, which breaks the encapsulation of the class. The chapter exercises challenge the student to develop classes for times, dates, rectangles and playing tic-tac-toe. Students generally enjoy game-playing programs. Mathematically inclined readers will enjoy the exercises on creating class Complex (for complex numbers), class Rational (for rational numbers) and class HugeInteger (for arbitrarily large integers).

Chapter 10, Classes: A Deeper Look, Part 2, continues the study of classes and presents additional object-oriented programming concepts. 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, which 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. In our discussion of const objects, we mention keyword mutable which is

used in a subtle manner to enable modification of "non-visible" implementation in `const` objects. We discuss dynamic memory allocation using `new` and `delete`. When `new` fails, the program terminates by default because `new` "throws an exception" in standard C++. We motivate the discussion of `static` class members with a video-game-based example. We emphasize how important it is to hide implementation details from clients of a class; then, we discuss proxy classes, which provide a means of hiding implementation (including the `private` data in class headers) from clients of a class. The chapter exercises include developing a savings-account class and a class for holding sets of integers.

Chapter 11, Operator Overloading; String and Array Objects, presents one of the most popular topics in our C++ courses. Students really enjoy this material. They find it a perfect complement to the detailed discussion of crafting valuable classes in Chapters 9 and 10. Operator overloading enables you 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 that 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 11, you'll learn how to "overload" the plus sign, so 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 `String`s. 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. Chapter 11 features a 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). Mathematically inclined students will enjoy creating the `polynomial` class in the exercises. This material is different from most programming languages and courses. Operator overloading is a complex topic, but an enriching one. Using operator overloading wisely helps you add extra "polish" to your classes. The discussions of class `Array` and class `String` are particularly valuable to students who have already used the C++ Standard Library `string` class and `vector` class template that provide similar capabilities. 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 10 exercises.

Chapter 12, Object-Oriented Programming: Inheritance, introduces one of the most fundamental capabilities of object-oriented programming languages—inheritance: 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. In the context of an **`Employee` hierarchy** case study, this substantially revised chapter presents a five-example sequence demonstrating `private` data, `protected` data and good software engineering with inheritance. We begin by demonstrating a class with `private` data members and `public` member functions to manipulate that data. Next, we implement a second class with additional capabilities, intentionally and tediously duplicating much of the first example's code. The third example begins our discussion of inheritance and software reuse—we use the class from the first example as a base class and quickly and simply inherit its data and functionality into a new derived class. This example introduces the inheritance mechanism and demonstrates that a derived class cannot access its base class's `private`

members directly. This motivates our fourth example, in which we introduce `protected` data in the base class and demonstrate that the derived class can indeed access the `protected` data inherited from the base class. The last example in the sequence demonstrates proper software engineering by defining the base class's data as `private` and using the base class's `public` member functions (that were inherited by the derived class) to manipulate the base class's `private` data in the derived class. 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 also compares inheritance (the *is-a* relationship) with composition (the *has-a* relationship) and introduces the *uses-a* and *knows-a* relationships.

Chapter 13, Object-Oriented Programming: Polymorphism, deals with another fundamental capability of object-oriented programming: polymorphic behavior. The completely revised Chapter 13 builds on the inheritance concepts presented in Chapter 12 and focuses on the relationships among classes in a class hierarchy and the powerful processing capabilities that these relationships enable. When many classes are related to a common base class through inheritance, each derived-class object may be treated as a base-class object. This enables programs to be written in a simple and 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 send a `draw` message to every object in a linked list of objects to be drawn. Each object knows how to draw itself. An object of a new class can be added to the program without modifying that program (as long as that new object also knows how to draw itself). The chapter discusses the mechanics of achieving polymorphic behavior via `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. We demonstrate abstract classes and polymorphic behavior by revisiting the **`Employee` hierarchy** of Chapter 12. We introduce an abstract `Employee` base class, from which classes `CommissionEmployee`, `HourlyEmployee` and `SalariedEmployee` inherit directly and class `BasePlusCommissionEmployee` inherits indirectly. In the past, our professional clients have insisted that we provide a deeper explanation that shows precisely how polymorphism is implemented in C++, and hence, precisely what execution time and memory "costs" are incurred when programming with this powerful capability. We responded by developing an illustration and a precise explanation of the *vtables* (`virtual` function tables) that the C++ compiler builds automatically to support polymorphism. To conclude, we introduce run-time type information (RTTI) and dynamic casting, which enable a program to determine an object's type at execution time, then act on that object accordingly. Using RTTI and dynamic casting, we give a 10% pay increase to employees of a specific type, then calculate the earnings for such employees. For all other employee types, we calculate their earnings polymorphically.

Chapter 14, Templates, discusses one of C++'s more powerful software reuse features, namely templates. Function templates and class templates enable you to specify, with a single code segment, an entire range of related overloaded functions (called function template specializations) or an entire range of related classes (called class-template special-

izations). This technique is called generic programming. Function templates were introduced in Chapter 6. This chapter presents additional discussions and examples on function template. We might write a single class template for a stack class, then have C++ generate separate class-template specializations, such as a "stack-of-`int`" class, a "stack-of-`float`" class, a "stack-of-`string`" class and so on. The chapter discusses using type parameters, nontype parameters and default types for class templates. We also discuss the relationships between templates and other C++ features, such as overloading, 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 in our discussion of the Standard Template Library (STL) containers, iterators and algorithms in Chapter 22.

Chapter 15, Stream Input/Output, contains a comprehensive treatment of standard C++ input/output capabilities. This chapter discusses a range of capabilities sufficient for performing most common I/O operations and overviews the remaining capabilities. Many of the I/O features are object oriented. This style of I/O makes use of other C++ features, such as references, function overloading and operator overloading. The various I/O capabilities of C++, including output with the stream insertion operator, input with the stream extraction operator, type-safe I/O, formatted I/O, unformatted I/O (for performance). Users can specify how to perform I/O for objects of user-defined types by overloading the stream insertion operator (<<) and the stream extraction operator (>>). This extensibility is one of C++'s most valuable features. C++ provides various stream manipulators that perform formatting tasks. This chapter discusses stream manipulators that provide capabilities such as displaying integers in various bases, controlling floating-point precision, setting field widths, displaying decimal point and trailing zeros, justifying output, setting and unsetting format state, setting the fill character in fields. We also present an example that creates user-defined output stream manipulators.

Chapter 16, Exception Handling, discusses how exception handling enables you to write programs that are robust, fault tolerant and appropriate for business-critical and mission-critical environments. The chapter discusses when exception handling is appropriate; introduces the basic capabilities of exception handling with `try` blocks, `throw` statements and `catch` handlers; 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. The exercises in this chapter show the student the diversity and power of C++'s exception-handling capabilities. We discuss rethrowing an exception, and we illustrate how `new` can fail when memory is exhausted. Many older C++ compilers return 0 by default when `new` fails. We show the new style of `new` failing by throwing a `bad_alloc` (bad allocation) exception. We illustrate how to use function `set_new_handler` to specify a custom function to be called to deal with memory-exhaustion situations. We discuss how to use the `auto_ptr` class template to `delete` dynamically allocated memory implicitly, thus avoiding memory leaks. To conclude this chapter, we present the Standard Library exception hierarchy.

Chapter 17, File Processing, discusses techniques for creating and processing both sequential files and random-access files. The chapter begins with an introduction to the data hierarchy from bits, to bytes, to fields, to records and to files. Next, we present the C++ view of files and streams. We discuss sequential files and build 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. We then discuss random-access files and build programs that show how to 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 case study combines the techniques of accessing files both sequentially and randomly into a complete transaction-processing program. Students in our industry seminars have mentioned 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 files and random-access files.

Chapter 18, Class `string` and String Stream Processing, The chapter discusses C++'s capabilities for inputting data from strings in memory and outputting data to strings in memory; these capabilities often are referred to as in-core formatting or string stream processing. Class `string` is a required component of the Standard Library. We preserved the treatment of C-like, pointer-based strings in Chapter 8 and later for several reasons. First, it strengthens your understanding of pointers. Second, 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 has accumulated over the last quarter of a century—this code processes strings as pointers, as does a large portion of the C++ code that has been written in industry over the last many years. In Chapter 18 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 in a selected `string` of characters. We show how to replace, erase and insert characters in a `string` and how to convert a `string` object to a C-style `char *` string.

Chapter 19, Searching and Sorting, discusses two of the most important classes of algorithms in computer science. We consider a variety of specific algorithms for each and compare them with regard to their memory consumption and processor consumption (introducing Big O notation, which indicates how hard an algorithm may have to work to solve a problem). Searching data involves determining whether a value (referred to as the search key) is present in the data and, if so, finding the value's location. In the examples and exercises of this chapter, we discuss a variety of searching algorithms, including: binary search and recursive versions of linear search and binary search. Through examples and exercises, Chapter 19 discusses the recursive merge sort, bubble sort, bucket sort and the recursive quicksort.

Chapter 20, 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, then 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 also helps the student master pointers. The chapter includes abundant examples that use 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. 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 preorder, 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. We include 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 17). Students then run the machine language produced by their compilers on the software simulators they built in the exercises of Chapter 8! The numerous exercises include 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, implementing an indexed-list class and a supermarket simulation that uses queueing. After studying Chapter 20, you are prepared for the treatment of STL containers, iterators and algorithms in Chapter 22. The STL containers are prepackaged, templatized data structures that most programmers will find sufficient for the vast majority of applications they will need to implement. The STL is a giant leap forward in achieving the vision of reuse.

Chapter 21, Bits, Characters, C Strings and `structs`, presents several important features. This chapter begins by comparing C++ structures to classes, then defining and using C-like structures. We show how to declare structures, initialize structures and pass structures to functions. The chapter features a high-performance card-shuffling and dealing simulation. This is an excellent opportunity for the instructor to emphasize the quality of algorithms. C++'s powerful bit-manipulation capabilities enable you to write programs that exercise lower-level hardware capabilities. This helps programs process bit strings, set individual bits 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 8 and presented the most popular string-manipulation functions. In Chapter 22, we continue our presentation of characters and C-style `char *` strings. We present the various character-manipulation capabilities of the `<cctype>` library—such as the ability to test a character to determine whether it is a digit, an alphabetic character, an alphanumeric character, a hexadecimal digit, a lowercase letter or an uppercase letter. 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. The numerous 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 spelling-checker program. 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.

Chapter 22, Standard Template Library (STL), Throughout this book, we discuss the importance of software reuse. Recognizing that many data structures and algorithms

are commonly used by C++ programmers, the C++ standard committee added the Standard Template Library (STL) to the C++ Standard Library. The STL defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures. The STL offers proof of concept for generic programming with templates—introduced in Chapter 14 and demonstrated in detail in Chapter 20. This chapter introduces the STL and discusses its three key components—containers (popular templatized data structures), iterators and algorithms. The STL containers are data structures capable of storing objects of any data type. We'll see that there are three container categories—first-class containers, adapters and near containers. STL iterators, which have similar properties to those of pointers, are used by programs to manipulate the STL-container elements. In fact, standard arrays can be manipulated as STL containers, using standard pointers as iterators. We'll see that manipulating containers with iterators is convenient and provides tremendous expressive power when combined with STL algorithms—in some cases, reducing many lines of code to a single statement. STL algorithms are functions that perform common data manipulations such as searching, sorting and comparing elements (or entire containers). There are approximately 70 algorithms implemented in the STL. Most of these use iterators to access container elements. We'll see that each first-class container supports specific iterator types, some of which are more powerful than others. A container's supported iterator type determines whether the container can be used with a specific algorithm. Iterators encapsulate the mechanism used to access container elements. This encapsulation enables many of the STL algorithms to be applied to several containers without regard for the underlying container implementation. As long as a container's iterators support the minimum requirements of the algorithm, then the algorithm can process that container's elements. This also enables you to create algorithms that can process the elements of multiple different container types. Chapter 20 discusses how to implement data structures with pointers, classes and dynamic memory. Pointer-based code is complex, and the slightest omission or oversight can lead to serious memory-access violations and memory-leak errors with no compiler complaints. Implementing additional data structures such as deques, priority queues, sets, maps, etc. requires substantial additional work. In addition, if many programmers on a large project implement similar containers and algorithms for different tasks, the code becomes difficult to modify, maintain and debug. An advantage of the STL is that you can reuse the STL containers, iterators and algorithms to implement common data representations and manipulations. This reuse results in substantial development-time and resource savings. This is a friendly, accessible chapter that should convince you of the value of the STL and encourage further study.

Chapter 23, Game Programming with Ogre, introduces game programming with 3D graphics using Ogre (Object-oriented Graphics Rendering Engine). Ogre is one of the leading open source graphics engines. It has been used in commercial applications including several computer games. We discuss the basic concepts of 3D game programming including graphics, 3D models, sound, user input, collision detection and controlling game speed. We provide a complete working example of a simple game featuring a play mechanic similar to the classic video game Pong®, originally developed by Atari in 1972. The chapter walks through the example, explaining key concepts and functions as they are encountered. We discuss the various resources used by Ogre and how to create them using scripts. Students will learn how to move, position and resize objects in a 3D

environment, perform simple collision detection, display text within the game and respond to user input from the keyboard. We also demonstrate how to use OgreAL, a wrapper for the OpenAL audio library, to add sound effects to the game.

Chapter 24, Boost Libraries, Technical Report 1 and C++0x, focuses on the future of C++. We introduce the Boost Libraries, a collection of free, open source C++ libraries. The Boost libraries are carefully designed to work well with the C++ Standard Library. We then discuss Technical Report 1 (TR1), a description of proposed changes and additions to the Standard Library. Many of the libraries in TR1 were derived from libraries currently in Boost. The chapter briefly describes all the libraries included in TR1. We provide in-depth code examples for two of the most useful libraries, `Boost.Regex` and `Boost.Smart_ptr`. The `Boost.Regex` library provides support for regular expressions. We demonstrate how to use the library to search a string for matches to a regular expression, validate data, replace parts of a string and split a string into tokens. The `Boost.Smart_ptr` library provides smart pointers to help manage dynamically allocated memory. We discuss the two types of smart pointers included in TR1—`shared_ptr` and `weak_ptr`. We provide examples to demonstrate how these can be used to avoid common memory management errors. This chapter also discusses the upcoming release of the new standard for C++, currently referred to as C++0x. We describe the goals for the new standard and overview the core language changes most likely to be standardized.

Chapter 25, Other Topics, is a collection of miscellaneous C++ topics. This chapter discusses one additional cast operator—`const_cast`. This operator, along with `static_cast` (Chapter 5), `dynamic_cast` (Chapter 13) and `reinterpret_cast` (Chapter 17), provide a more robust mechanism for converting between types than do the original cast operators C++ inherited from C (which are now deprecated). We discuss namespaces, a feature particularly crucial for software developers who build substantial systems, especially for those who build systems from class libraries. Namespaces prevent naming collisions, which can hinder such large software efforts. The Chapter discusses the operator keywords, which are useful for programmers who have keyboards that do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`. These keywords can also be used by programmers who do not like cryptic operator symbols. We discuss keyword `mutable`, which allows a member of a `const` object to be changed. Previously, this was accomplished by "casting away `const`-ness", which is considered a dangerous practice. We also discuss pointer-to-member operators `.*` and `->*`, multiple inheritance (including the problem of "diamond inheritance") and `virtual` base classes.

Appendix A, Operator Precedence and Associativity Chart, presents the complete set of C++ operator symbols, in which each operator appears on a line by itself with its operator symbol, its name and its associativity.

Appendix B, ASCII Character Set. All the programs in this book use the ASCII character set, which is presented in this appendix.

Appendix C, Fundamental Types, lists C++'s fundamental types.

Appendix D, 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 E, C Legacy Code Topics, presents additional material 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) and append the output of a program to an existing file. We develop functions that use variable-length argument lists and show how to pass command-line arguments to function main and use them in a program. We discuss how to compile programs whose components are spread across multiple files, register functions with atexit to be executed at program termination and terminate program execution with function exit. We also discuss the const and volatile type qualifiers, specifying the type of a numeric constant using the integer and floating-point suffixes, using the signal-handling library to trap unexpected events, creating and using dynamic arrays with calloc and realloc, using unions as a space-saving technique and using linkage specifications when C++ programs are to be linked with legacy C code. As the title suggests, this appendix is intended primarily for C++ programmers who will be working with C legacy code as most C++ programmers are almost certain to do at one point in their careers.

Appendix F, Preprocessor, discusses the preprocessor's directives. The appendix includes more complete information on the #include directive, which 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 appendix explains conditional compilation, which enables you 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__, __STDC__, __TIME__ and __TIMESTAMP__) are presented. Finally, macro assert of the header file <cassert> is discussed, which is valuable in program testing, debugging, verification and validation.

Appendix G, ATM Case Study Code, contains the implementation of our case study on object-oriented design with the UML. This appendix is discussed in the overview of the case study (presented shortly).

Appendix H, UML 2: Additional Diagram Types, overviews the UML 2 diagram types that are not found in the OOD/UML Case Study.

Appendix I, Using the Visual Studio Debugger, demonstrates key features of the Visual Studio Debugger, which allows a programmer to monitor the execution of applications to locate and remove logic errors. The appendix presents step-by-step instructions, so students learn how to use the debugger in a hands-on manner.

Appendix J, Using the GNU C++ Debugger, demonstrates key features of the GNU C++ Debugger, which allows a programmer to monitor the execution of applications to locate and remove logic errors. The appendix presents step-by-step instructions, so students learn how to use the debugger in a hands-on manner.

Bibliography. The Bibliography lists many books and articles to encourage the student to do further reading on C++ and OOP.

Index. The comprehensive index enables you to locate by keyword any term or concept throughout the text.

Object-Oriented Design of an ATM with the UML: A Tour of the Optional Software Engineering Case Study

In this section, we tour the book's optional case study of object-oriented design with the UML. This tour previews the contents of the nine Software Engineering Case Study sec-

tions (in Chapters 1–7, 9 and 13). After completing this case study, you'll be thoroughly familiar with a carefully reviewed object-oriented design and implementation for a significant C++ application.

The design presented in the ATM case study was developed at Deitel & Associates, Inc. and scrutinized by a distinguished developmental review team of industry professionals and academics. We crafted this design to meet the requirements of introductory course sequences. Real ATM systems used by banks and their customers worldwide are based on more sophisticated designs that take into consideration many more issues than we have addressed here. Our primary goal throughout the design process was to create a simple design that would be clear to OOD and UML novices, while still demonstrating key OOD concepts and the related UML modeling techniques. We worked hard to keep the design and the code relatively small so that it would work well in the introductory course sequence.

Section 1.21, Software Engineering Case Study: Introduction to Object Technology and the UML—introduces the object-oriented design case study with the UML. The section introduces the basic concepts and terminology of object technology, including classes, objects, encapsulation, inheritance and polymorphism. We discuss the history of the UML. This is the only required section of the case study.

Section 2.8, (Optional) Software Engineering Case Study: Examining the ATM Requirements Specification—discusses a *requirements specification* that specifies the requirements for a system that we'll design and implement—the software for a simple automated teller machine (ATM). We investigate the structure and behavior of object-oriented systems in general. We discuss how the UML will facilitate the design process in subsequent Software Engineering Case Study sections by providing several additional types of diagrams to model our system. We include a list of URLs and book references on object-oriented design with the UML. We discuss the interaction between the ATM system specified by the requirements specification and its user. Specifically, we investigate the scenarios that may occur between the user and the system itself—these are called *use cases*. We model these interactions, using *use case diagrams* of the UML.

Section 3.11, (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Specification—begins to design the ATM system. We identify its classes, or "building blocks," by extracting the nouns and noun phrases from the requirements specification. We arrange these classes into a UML class diagram that describes the class structure of our simulation. The class diagram also describes relationships, known as *associations*, among classes.

Section 4.13, (Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM System—focuses on the attributes of the classes discussed in Section 3.11. A class contains both *attributes* (data) and *operations* (behaviors). As we'll see in later sections, changes in an object's attributes often affect the object's behavior. To determine the attributes for the classes in our case study, we extract the adjectives describing the nouns and noun phrases (which defined our classes) from the requirements specification, then place the attributes in the class diagram we created in Section 3.11.

Section 5.11, (Optional) Software Engineering Case Study: Identifying Objects' States and Activities in the ATM System—discusses how an object, at any given time, occupies a specific condition called a *state*. A *state transition* occurs when that object receives a message to change state. The UML provides the *state machine diagram*, which

identifies the set of possible states that an object may occupy and models that object's state transitions. An object also has an *activity*—the work it performs in its lifetime. The UML provides the *activity diagram*—a flowchart that models an object's activity. In this section, we use both types of diagrams to begin modeling specific behavioral aspects of our ATM system, such as how the ATM carries out a withdrawal transaction and how the ATM responds when the user is authenticated.

Section 6.22, (Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System—identifies the operations, or services, of our classes. We extract from the requirements specification the verbs and verb phrases that specify the operations for each class. We then modify the class diagram of Section 3.11 to include each operation with its associated class. At this point in the case study, we will have gathered all information possible from the requirements specification. However, as future chapters introduce such topics as inheritance, we'll modify our classes and diagrams.

Section 7.12, (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System—provides a "rough sketch" of the model for our ATM system. In this section, we see how it works. We investigate the behavior of the simulation by discussing *collaborations*—messages that objects send to each other to communicate. The class operations that we discovered in Section 6.22 turn out to be the collaborations among the objects in our system. We determine the collaborations, then collect them into a *communication diagram*—the UML diagram for modeling collaborations. This diagram reveals which objects collaborate and when. We present a communication diagram of the collaborations among objects to perform an ATM balance inquiry. We then present the UML *sequence diagram* for modeling interactions in a system. This diagram emphasizes the chronological ordering of messages. A sequence diagram models how objects in the system interact to carry out withdrawal and deposit transactions.

Section 9.11, (Optional) Software Engineering Case Study: Starting to Program the Classes of the ATM System—takes a break from designing the system's behavior. We begin the implementation process to emphasize the material discussed in Chapter 9. Using the UML class diagram of Section 3.11 and the attributes and operations discussed in Section 4.13 and Section 6.22, we show how to implement a class in C++ from a design. We do not implement all classes—because we have not completed the design process. Working from our UML diagrams, we create code for the `Withdrawal` class.

Section 13.10, (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System—continues our discussion of object-oriented programming. We consider inheritance—classes sharing common characteristics may inherit attributes and operations from a "base" class. In this section, we investigate how our ATM system can benefit from using inheritance. We document our discoveries in a class diagram that models inheritance relationships—the UML refers to these relationships as *generalizations*. We modify the class diagram of Section 3.11 by using inheritance to group classes with similar characteristics. This section concludes the design of the model portion of our simulation. We fully implement this model in 877 lines of C++ code in Appendix G.

Appendix G, ATM Case Study Code—The majority of the case study involves designing the model (i.e., the data and logic) of the ATM system. In this appendix, we implement that model in C++. Using all the UML diagrams we created, we present the C++ classes necessary to implement the model. We apply the concepts of object-oriented design with the UML and object-oriented programming in C++ that you learned in the

chapters. By the end of this appendix, students will have completed the design and implementation of a real-world system, and should feel confident tackling larger systems, such as those that professional software engineers build.

Appendix H, UML 2: Additional Diagram Types—Overviews the UML 2 diagram types that are not found in the OOD/UML Case Study.

Student Resources Included with C++ *How to Program, 6/e*

Many C++ development tools are available. We wrote *C++ How to Program, 6/e* primarily using Microsoft's free Visual C++ Express Edition (which is included on the CD that accompanies this book) and the free GNU C++, which is already installed on most Linux systems and can be installed on Mac OS X systems as well. You can learn more about Visual C++ Express at `msdn.microsoft.com/vstudio/express/visualc`. You can learn more about GNU C++ at `gcc.gnu.org`. Apple includes GNU C++ in their Xcode development tools, which Max OS X users can download from `developer.apple.com/tools/xcode`.

Additional resources and software downloads are available in our C++ Resource Center:

```
www.deitel.com/cplusplus/
```

and at the website for this book:

```
www.deitel.com/books/cppphtp6/
```

For a list of other compilers that are available free for download, visit the following sites:

```
www.thefreecountry.com/developercity/ccompilers.shtml
www.compilers.net
```

Warnings and Error Messages on Older C++ Compilers

The programs in this book are designed to be used with compilers that support standard C++. However, there are variations among compilers that may cause occasional warnings or errors. In addition, though the standard specifies various situations that require errors to be generated, it does not specify the messages that compilers should issue. Warnings and error messages vary among compilers—this is normal.

Some older C++ compilers, such as Microsoft Visual C++ 6, Borland C++ 5.5 and various earlier versions of GNU C++ generate error or warning messages in places where newer compilers do not. Although most of the examples in this book will work with these older compilers, there are a few examples that need minor modifications to work with older compilers.

Notes Regarding `using` Declarations and C Standard Library Functions

The C++ Standard Library includes the functions from the C Standard Library. According to the C++ standard document, the contents of the header files that come from the C Standard Library are part of the "`std`" namespace. Some compilers (old and new) generate error messages when `using` declarations are encountered for C functions.

C++ Multimedia Cyber Classroom, 6/e

C++ How to Program, 6/e includes a free, web-based, audio-intensive interactive multimedia ancillary to the book—*The C++ Multimedia Cyber Classroom, 6/e*—available with new

books purchased from Prentice Hall. If you have a used book, you can purchase standalone access to the *Cyber Classroom* at MyPearsonStore.com. Our Web-based *Cyber Classroom* includes audio walkthroughs of code examples in Chapters 1–14, solutions to about half of the exercises in the book, a lab manual and more. For more information about the web-based *Cyber Classroom*, please visit

    www.prenhall.com/deitel/cyberclassroom/

Students who use our Cyber Classrooms like its interactivity and reference capabilities. Professors tell us that their students enjoy using the Cyber Classroom and consequently spend more time on the courses, mastering more of the material than in textbook-only courses.

Instructor Resources for C++ *How to Program, 6/e*

C++ How to Program, 6/e has extensive instructor resources. The Prentice Hall *Instructor's Resource Center* contains the *Solutions Manual* with solutions to the vast majority of the end-of-chapter exercises, a *Test Item File* of multiple-choice questions (approximately two per book section) and PowerPoint® slides containing all the code and figures in the text, plus bulleted items that summarize the key points in the text. Instructors can customize the slides. If you are not already a registered faculty member, contact your Prentice Hall representative or visit vig.prenhall.com/replocator/.

Deitel® Buzz Online Free E-mail Newsletter

Each week, the *Deitel® Buzz Online* newsletter announces our latest Resource Center(s) and includes commentary on industry trends and developments, links to free articles and resources from our published books and upcoming publications, product-release schedules, errata, challenges, anecdotes, information on our corporate instructor-led training courses and more. It's also a good way for you to keep posted about issues related to *C++ How to Program, 6/e*. To subscribe, visit

    www.deitel.com/newsletter/subscribe.html

The Deitel Online Resource Centers

Our website www.deitel.com provides Resource Centers on various topics including programming languages, software, Web 2.0, Internet business and open source projects (Fig. 2). The Resource Centers evolved out of the research we've done for our books and business endeavors. We've found many exceptional resources including tutorials, documentation, software downloads, articles, blogs, videos, code samples, books, e-books and more. Most of them are free. In the spirit of Web 2.0, we share these resources with the worldwide community. The Deitel Resource Centers are a starting point for your own research. We help you wade through the vast amount of content on the Internet by providing links to the most valuable resources. Each week we announce our latest Resource Centers in our newsletter, the *Deitel® Buzz Online* (www.deitel.com/newsletter/subscribe.html). The following Resource Centers may be of interest to you as you study *C++ How to Program, 6/e*:

- C++
- C++ Boost Libraries

Deitel Resource Centers		
Programming	Linux	Skype
.NET	MySQL	Social Media
.NET 3.0	Open Source	Social Networking
Ajax	Search Engines	Software as a Service (SaaS)
Apex On-Demand Programming Language	Wikis	Virtual Worlds
ASP.NET	Windows Vista	Web 2.0
C		Web 3.0
C#	***Microsoft***	Widgets
C++	.NET	Wikis
C++ Boost Libraries	.NET 3.0	
C++ Game Programming	ASP.NET	***Internet Business***
Code Search Engines and Code Sites	C#	Affiliate Programs
Computer Game Programming	DotNetNuke (DNN)	Google Adsense
CSS 2.1	Internet Explorer 7	Google Analytics
Flash 9	Silverlight	Google Services
Flex	Visual Basic	Internet Advertising
Java	Visual C++	Internet Business Initiative
Java Certification and Assessment Testing	Windows Vista	Internet Public Relations
Java Design Patterns		Link Building
Java EE 5	***Java***	Podcasting
Java SE 6	Java	Search Engine Optimization
JavaFX	Java Certification and Assessment Testing	Sitemaps
JavaScript	Java Design Patterns	Web Analytics
OpenGL	Java EE 5	Website Monetization
Perl	Java SE 6	
PHP	JavaFX	***Open Source***
Programming Projects	JavaScript	Apache
Python		DotNetNuke (DNN)
Ruby	***Web 2.0***	Eclipse
Silverlight	Alert Services	Firefox
Visual Basic	Attention Economy	Linux
Visual C++	Blogging	MySQL
Web Services	Building Web Communities	Open Source
Web 3D Technologies	Community Generated Content	Perl
XML	Google Base	PHP
	Google Video	Python
Software	Google Web Toolkit	Ruby
Apache	Internet Video	
DotNetNuke (DNN)	Joost	***Other Topics***
Eclipse	Mashups	Computer Games
Firefox	Microformats	Computing Jobs
Internet Explorer 7	Ning	Gadgets and Gizmos
	Recommender Systems	Sudoku
	RSS	

Fig. 2 | Deitel Resource Centers.

- C++ Game Programming
- Code Search Engines and Code Sites
- Computer Game Programming
- Computing Jobs
- Open Source
- Programming Projects
- Eclipse
- Linux
- .NET
- Windows Vista

Acknowledgments

It is a great pleasure to acknowledge 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 people at Deitel & Associates, Inc. devoted long hours to this project—thanks especially to Abbey Deitel and Barbara Deitel.

We'd also like to thank one of the participants in our Honors Internship program who contributed to this publication—Greg Ayer, a computer science major at Northeastern University.

We are fortunate to have worked on this project with the talented and dedicated team of publishing professionals at Prentice Hall. We appreciate the extraordinary efforts of Marcia Horton, Editorial Director of Prentice Hall's Engineering and Computer Science Division. Carole Snyder and Dolores Mars did an extraordinary job recruiting the book's review team and managing the review process. Francesco Santalucia (an independent artist) and Kristine Carney of Prentice Hall did a wonderful job designing the book's cover; we provided the concept, and they made it happen. Vince O'Brien, Scott Disanno, Bob Engelhardt and Marta Samsel did a marvelous job managing the book's production.

We wish to acknowledge the efforts of our reviewers. Adhering to a tight time schedule, they scrutinized the text and the programs, providing countless suggestions for improving the accuracy and completeness of the presentation.

We sincerely appreciate the efforts of our fifth edition post-publication reviewers and our sixth edition reviewers:

C++ How to Program 6/e Reviewers

Academic and Industry Reviewers: Dr. Richard Albright (Goldey-Beacom College), William B. Higdon (University of Indianapolis), Howard Hinnant (Apple), Anne B. Horton (Lockheed Martin), Terrell Hull (Logicalis Integration Solutions), Rex Jaeschke (Independent Consultant), Maria Jump (The University of Texas at Austin), Geoffrey S. Knauth (GNU), Don Kostuch (Independent Consultant), Colin Laplace (Freelance Software Consultant), Stephan T. Lavavej (Microsoft), Amar Raheja (California State Polytechnic University, Pomona), G. Anthony Reina (University of Maryland University College, Europe), Daveed Vandevoorde (C++ Standards Committee), Jeffrey Wiener (DEKA Research & Development Corporation, New Hampshire Community Technical College), and Chad Willwerth (University of Washington, Tacoma). **Ogre Reviewers:** Casey Bor-

ders (Sensis Corp.), Gregory Junker (Author of *Pro OGRE3D Programming*, Apress Books), Mark Pope (THQ, Inc.), and Steve Streeting (Torus Knot Software, Ltd.). **Boost/C++OX Reviewers:** Edward Brey (Kohler Co.), Jeff Garland (Boost.org), Douglas Gregor (Indiana University), and Björn Karlsson (Author of *Beyond the C++ Standard Library: An Introduction to Boost*, Addison-Wesley/Readsoft, Inc.).

C++ How to Program 5/e Reviewers
Academic Reviewers: Richard Albright (Goldey-Beacom College), Karen Arlien (Bismarck State College), David Branigan (DeVry University, Illinois), Jimmy Chen (Salt Lake Community College), Martin Dulberg (North Carolina State University), Ric Heishman (Northern Virginia Community College), Richard Holladay (San Diego Mesa College), William Honig (Loyola University), Earl LaBatt (OPNET Technologies, Inc./University of New Hampshire), Brian Larson (Modesto Junior College), Robert Myers (Florida State University), Gavin Osborne (Saskatchewan Institute of Applied Science and Technology), Wolfgang Pelz (The University of Akron), and Donna Reese (Mississippi State University). **Industry Reviewers:** Curtis Green (Boeing Integrated Defense Systems), Mahesh Hariharan (Microsoft), James Huddleston (Independent Consultant), Ed James-Beckham (Borland Software Corporation), Don Kostuch (Independent Consultant), Meng Lee (Hewlett-Packard), Kriang Lerdsuwanakij (Siemens Limited), William Mike Miller (Edison Design Group, Inc.), Mark Schimmel (Borland International), Vicki Scott (Metrowerks), James Snell (Boeing Integrated Defense Systems), and Raymond Stephenson (Microsoft). **OOD/UML Optional Software Engineering Case Study Reviewers:** Sinan Si Alhir (Independent Consultant), Karen Arlien (Bismarck State College), David Branigan (DeVry University, Illinois), Martin Dulberg (North Carolina State University), Ric Heishman (Northern Virginia Community College), Richard Holladay (San Diego Mesa College), Earl LaBatt (OPNET Technologies, Inc./University of New Hampshire), Brian Larson (Modesto Junior College), Gavin Osborne (Saskatchewan Institute of Applied Science and Technology), Praveen Sadhu (Infodat International, Inc.), Cameron Skinner (Embarcadero Technologies, Inc./OMG), and Steve Tockey (Construx Software).

These reviewers scrutinized every aspect of the text and made countless suggestions for improving the accuracy and completeness of the presentation.

Well, there you have it! Welcome to the exciting world of C++ and object-oriented programming. We hope you enjoy this look at contemporary computer programming. Good luck! As you read the book, we would sincerely appreciate your comments, criticisms, corrections and suggestions for improving the text. Please address all correspondence to:

`deitel@deitel.com`

We'll respond promptly, and post corrections and clarifications on:

`www.deitel.com/books/cpphtp6/`

We hope you enjoy reading *C++ How to Program, Sixth Edition* as much as we enjoyed writing it!

Paul J. Deitel
Dr. Harvey M. Deitel
Maynard, Massachusetts
July 2007

About the Authors

Paul J. Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT's Sloan School of Management, where he studied Information Technology. He holds the Java Certified Programmer and Java Certified Developer certifications, and has been designated by Sun Microsystems as a Java Champion. Through Deitel & Associates, Inc., he has delivered Java, C, C++, C# and Visual Basic courses to industry clients, including IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Stratus, Cambridge Technology Partners, Open Environment Corporation, One Wave, Hyperion Software, Adra Systems, Entergy, CableData Systems, Nortel Networks, Puma, iRobot, Invensys and many more. He has also lectured on Java and C++ for the Boston Chapter of the Association for Computing Machinery. He and his father, Dr. Harvey M. Deitel, are the world's best-selling programming language textbook authors.

Dr. Harvey M. Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 45 years of academic and industry experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees from the MIT and a Ph.D. from Boston University. 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 his son, Paul J. Deitel. He and Paul are the co-authors of several dozen books and multimedia packages and they are writing many more. With translations published in Japanese, German, Russian, Spanish, Traditional Chinese, Simplified Chinese, Korean, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish, the Deitels' texts have earned international recognition. Dr. Deitel has delivered hundreds of professional seminars to major corporations, academic institutions, government organizations and the military.

About Deitel & Associates, Inc.

Deitel & Associates, Inc., is an internationally recognized corporate training and content-creation organization specializing in computer programming languages, Internet and World Wide Web software technology, object technology education and Internet business development through its Internet Business Initiative. The company provides instructor-led courses on major programming languages and platforms, such as C++, Java, Advanced Java, C, C#, Visual C++, Visual Basic, XML, Perl, Python, object technology and Internet and World Wide Web programming. The founders of Deitel & Associates, Inc., are Dr. Harvey M. Deitel and Paul J. Deitel. The company's clients include many of the world's largest companies, government agencies, branches of the military, and academic institutions. Through its 30-year publishing partnership with Prentice Hall, Deitel & Associates, Inc. publishes leading-edge programming textbooks, professional books, interactive multimedia *Cyber Classrooms*, *Complete Training Courses*, Web-based training courses and e-content for the popular course management systems WebCT, Blackboard and Pearson's CourseCompass. Deitel & Associates, Inc., and the authors can be reached via e-mail at:

`deitel@deitel.com`

To learn more about Deitel & Associates, Inc., its publications and its worldwide *Dive Into*® Series Corporate Training curriculum, visit:

```
www.deitel.com
```

and subscribe to the free *Deitel*® *Buzz Online* e-mail newsletter at:

```
www.deitel.com/newsletter/subscribe.html
```

Check out the growing list of online Deitel Resource Centers at:

```
www.deitel.com/resourcecenters.html
```

Individuals wishing to purchase Deitel publications can do so through:

```
www.deitel.com/books/index.html
```

Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Prentice Hall. For more information, visit

```
www.prenhall.com/mischtm/support.html#order
```