Preface

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

Welcome to *C++ for Programmers*! At Deitel & Associates, we write programming language professional books and textbooks for publication by Prentice Hall, deliver programming languages corporate training courses at organizations worldwide and develop Internet businesses. This book is intended for programmers who do not yet know C++, and may or may not know object-oriented programming.

Features of *C++ for Programmers*

The Tour of the Book section of this Preface will give you a sense of *C++ for Programmers'* coverage of C++ and object-oriented programming. Here's some key features of the book:

- **Early Classes and Objects Approach.** We present object-oriented programming, where appropriate, from the start and throughout the text.
- **Integrated Case Studies.** We develop 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 E.
- **Unified Modeling Language™ 2 (UML 2).** The Unified Modeling Language (UML) has become the preferred graphical modeling language for designers of object-oriented systems. 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 emphasize the UML in the optional OOD/UML ATM case study
- **Optional OOD/UML ATM Case Study.** We introduce a concise subset of the UML 2, then guide you through a first design experience intended for the novice object-oriented designer/programmer. The case study was reviewed by a distinguished team of OOD/UML industry professionals and academics. The case study is not an exercise; rather, it's 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.
- **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.

- **Class `string`.** We use class `string` instead of C-like pointer-based `char *` strings for most string manipulations throughout the book. We include discussions of `char *` strings in Chapters 8, 10, 11 and 19 to give you practice with pointer manipulations, to illustrate dynamic memory allocation with `new` and `delete`, to build our own `String` class, and to prepare you 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 you for working with C and C++ legacy code and to use as a basis for building our own customized `Array` class in Chapter 11.
- **Treatment of Inheritance and Polymorphism.** Chapters 12–13 include an `Employee` class hierarchy that makes the treatment of inheritance and polymorphism clear and accessible for programmers 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 you a solid understanding of how these capabilities really work. More importantly, it helps you appreciate the overhead of polymorphism—in terms of additional memory consumption and processor time. This helps you 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 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 20 introduces the STL and discusses its three key components—containers, iterators and algorithms. Using STL components provides tremendous expressive power, often reducing many lines of non-STL 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:* A PDF copy of the C++ standard (document number INCITS/ISO/IEC 14882-2003) can be purchased at `webstore.ansi.org/ansidocstore/default.asp`.]
- **Future of C++.** In Chapter 21, 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.

- **Debugger Appendices.** We include two Using the Debugger appendices—Appendix G, Using the Visual Studio Debugger, and Appendix H, Using the GNU C++ Debugger.
- **Code Testing on Multiple Platforms.** We tested the code examples on various popular C++ platforms. For the most part, the book's examples port easily to 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 was carefully reviewed by distinguished industry developers and academics. We believe that this book will provide you with an informative, interesting, challenging and entertaining C++ educational 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.

Learning Features

C++ for Programmers 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 programming languages in industry classrooms worldwide. The Deitels have taught courses at all levels to government, industry, military and academic clients of Deitel & Associates.

Live-Code Approach. *C++ for Programmers* 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.

Syntax Shading. We syntax-shade all the C++ code, similar to the way most C++ integrated development environments (IDEs) and code editors syntax-color code. This greatly improves code readability—an especially important goal, given that this book contains over 15,500 lines of code. Our syntax-shading conventions are as follows:

```
comments appear in italic
keywords appear in bold italic
errors and ASP.NET script delimiters appear in bold black
constants and literal values appear in bold gray
all other code appears in plain black
```

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

Using Fonts for Emphasis. We place the key terms and the index's page reference for each defining occurrence in ***bold italic*** 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++ for Programmers* are available for download from `www.deitel.com/books/cppfp/`.

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've 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 outlines help 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've gleaned from a combined seven decades of programming experience—they provide a basis on which to build good software.

Good Programming Practice

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

Common Programming Error

Pointing out these Common Programming Errors *reduces the likelihood that you'll make the same mistakes.*

Error-Prevention Tip

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 Tip

These tips highlight opportunities for making your programs run faster or minimizing the amount of memory that they occupy.

Portability Tip

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 Observation

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.

Thousands of Index Entries. We've 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++ for Programmers*. 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.

Chapter 1, Introduction, discusses the origin of the C++ programming language, and introduces a typical C++ programming environment. We walk through a "test drive" of a typical C++ application on the Windows and Linux platforms. We also introduce 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 C++. The programs in this chapter illustrate how to display data on the screen, obtain data from the keyboard, make decisions and perform arithmetic operations.

Chapter 3, Introduction to Classes and Objects, provides a friendly early introduction to classes and objects. 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++. 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 an instructor 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. 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. We explain 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 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 principle of good software engineering—separating interface from implementation.

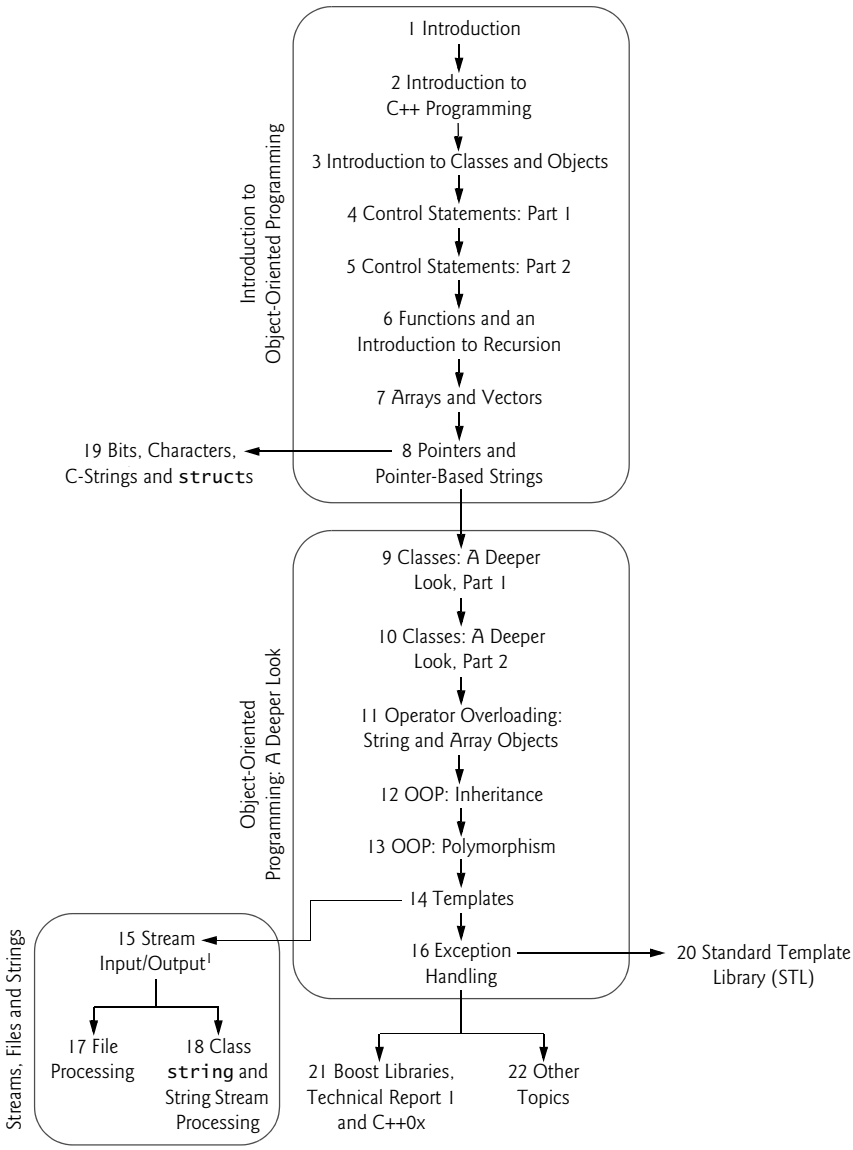


Fig. 1 | *C++ for Programmers* chapter dependency chart.

Chapter 4, Control Statements: Part 1, focuses on the program-development process involved in creating useful classes. The chapter introduces some 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. 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. The chapter also a discusses 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 you can build your own functions. The chapter's first example continues the **`GradeBook` class case study** with an example of a function with multiple parameters. You may 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. We discuss the function call stack and activation records to explain how C++ keeps 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.

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. The second version uses a two-dimensional array to store the grades of a number of students on multiple exams in a semester. Another key feature of this chapter is the discussion of elementary sorting and searching techniques.

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). We frequently use string objects (introduced in Chapter 3) 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. In C and "raw C++" arrays and strings are pointers to array and string contents in memory (even function names are pointers).

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. We discuss 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.

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 scenario. 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.

Chapter 11, Operator Overloading; String and Array Objects, presents one of the most popular topics in our C++ courses. Professionals 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 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 see how to "overload" the plus sign, so when it is written between two String objects in an expres-

sion, 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 and a `Date` class. Using operator overloading wisely helps you add extra "polish" to your classes.

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 chapter presents a five-example sequence demonstrating `private` data, `protected` data and good software engineering with inheritance. 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. 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. 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 include 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.

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 specializations). This technique is called generic programming. 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. We greatly enhance

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

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. 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 (>>). 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. We discuss rethrowing an exception, and 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.

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, Bits, Characters, C Strings and `structs`, 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. 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. We discuss C-style `char *` string manipulation in Chapter 8, where we present the most popular string-manipulation functions. In Chapter 19, 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.

Chapter 20, Standard Template Library (STL), discusses the STL's 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. This chapter discusses the STL's three key components—containers (templatized data structures), iterators and algorithms. Containers are data structures capable of storing objects of any type. We'll see that there are three container categories—first-class containers, adapters and near containers. Iterators, which have similar properties to those of pointers, are used by programs to manipulate the container elements. In fact, standard arrays can be manipulated as STL containers, using pointers as iterators. 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). Most of these use iterators to access container elements.

Chapter 21, 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 the TR1 libraries. 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++.

Chapter 22, Other Topics, is a collection of miscellaneous C++ topics. We discuss one more cast operator—`const_cast`. This operator, `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. Namespaces prevent naming collisions, which can hinder such large software efforts. 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, 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 E, ATM Case Study Code, contains the implementation of our case study on object-oriented design with the UML. This appendix is discussed in the tour of the case study (presented shortly).

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

Appendix G, 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 you learn how to use the debugger in a hands-on manner.

Appendix H, Using the GNU C++ Debugger, demonstrates key features of the GNU C++ Debugger. The appendix presents step-by-step instructions, so you learn how to use the debugger in a hands-on manner.

Bibliography. The Bibliography lists many books and articles for further reading on C++ and object-oriented programming.

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 sections (in Chapters 1–7, 9 and 13). After completing this case study, you'll be thoroughly familiar with a carefully developed and 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. 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.

Section 1.10, 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 2.7, (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 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 4.11, (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 5.10, (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 4.11 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 E.

Appendix E, 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, you'll have completed the design and implementation of a real-world system, and should feel confident tackling larger systems.

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

Compilers and Other Resources

Many C++ development tools are available. We wrote *C++ for Programmers* primarily using Microsoft's free Visual C++ Express Edition (`www.microsoft.com/express/vc/`) and the free GNU C++ at `gcc.gnu.org`, which is already installed on most Linux systems and can be installed on Mac OS X systems as well. 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/cppfp/
```

For a list of other C++ compilers that are available free for download, visit:

```
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.

Some older C++ compilers 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 Stan-

dard Library are part of the "std" namespace. Some compilers (old and new) generate error messages when using declarations are encountered for C functions.

The Deitel Online Resource Centers

Our website provides Resource Centers (www.deitel.com/ResourceCenters.html) on various topics including programming languages, software, Web 2.0, Internet business and open source projects. The Resource Centers evolve out of the research we do for our books and business endeavors. We've found many (mostly free) exceptional resources including tutorials, documentation, software downloads, articles, blogs, videos, code samples, books, e-books and more. 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 read *C++ for Programmers*:

- C++
- Visual C++ 2008
- C++ Boost Libraries
- C++ Game Programming
- Code Search Engines and Code Sites
- Computer Game Programming
- Computing Jobs
- Open Source
- Programming Projects
- Eclipse
- Linux
- .NET
- Windows Vista

Deitel® Buzz Online Free E-mail Newsletter

Each week, the *Deitel® Buzz Online* newsletter announces our latest Resource Centers 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++ for Programmers*. To subscribe, visit

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

Deitel® *LiveLessons* Self-Paced Video Training

The Deitel® *LiveLessons* products are self-paced video training. Each collection provides approximately 14+ hours of an instructor guiding you through programming training.

Your instructor, Paul Deitel, has personally taught programming at organizations ranging from IBM to Sun Microsystems to NASA. With the powerful videos included in our *LiveLessons* products, you'll learn at your own pace as Paul guides you through programming fundamentals, object-oriented programming and additional topics.

Deitel® *LiveLessons* products are based on its corresponding best-selling books and Paul's extensive experience presenting hundreds corporate training seminars. To view sample videos, visit

```
www.deitel.com/books/livelessons/
```

The *Java Fundamentals I and II LiveLessons* are available now. For announcements about upcoming Deitel *LiveLessons* products, including *C++ Fundamentals*, *C# 2008 Fundamentals* and *JavaScript Fundamentals*, subscribe to the *Deitel® Buzz Online* email newsletter at `www.deitel.com/newsletter/subscribe.html`.

Deitel® *Dive-Into®* Series Instructor-Led Training

With our corporate, on-site, instructor-led *Dive-Into® Series* programming training courses (Fig. 2), professionals can learn C++, Java, C, Visual Basic, Visual C#, Visual C++, Python, and Internet and web programming from the internationally recognized professionals at Deitel & Associates, Inc. Our authors, teaching staff and contract instructors have taught over 1,000,000 people in more than 100 countries how to program in almost every major programming language through:

- *Deitel Developer Series* professional books
- *How to Program Series* textbooks
- University teaching
- Professional seminars
- Interactive multimedia CD-ROM Cyber Classrooms, Complete Training Courses and *LiveLessons* Video Training
- Satellite broadcasts

We're uniquely qualified to turn non-programmers into programmers and to help professional programmers move to new programming languages. For more information about our on-site, instructor-led *Dive-Into®* Series programming training, visit

`www.deitel.com/training/`

Deitel *Dive Into®* Series Programming Training Courses
Java
Intro to Java for Non-Programmers: Part 1
Intro to Java for Non-Programmers: Part 2
Java for Visual Basic, C or COBOL Programmers
Java for C++ or C# Programmers
Advanced Java
C++
Intro to C++ for Non-Programmers: Part 1
Intro to C++ for Non-Programmers: Part 2
C++ and Object Oriented Programming
C
Intro to C for Non-Programmers: Part 1
Intro to C for Non-Programmers: Part 2
C for Programmers

Fig. 2 | Deitel *Dive Into®* Series programming training courses. (Part 1 of 2.)

Deitel *Dive Into*® Series Programming Training Courses

Visual C# 2008

Intro to Visual C# 2008 for Non-Programmers: Part 1
Intro to Visual C# 2008 for Non-Programmers: Part 2
Visual C# 2008 for Visual Basic, C or COBOL Programmers
Visual C# 2008 for Java or C++ Programmers
Advanced Visual C# 2008

Visual Basic 2008

Intro to Visual Basic 2008 for Non-Programmers: Part 1
Intro to Visual Basic 2008 for Non-Programmers: Part 2
Visual Basic 2008 for VB6, C or COBOL Programmers
Visual Basic 2008 for Java, C# or C++ Programmers
Advanced Visual Basic 2008

Visual C++ 2008

Intro to Visual C++ 2008 for Non-Programmers: Part 1
Intro to Visual C++ 2008 for Non-Programmers: Part 2
Visual C++ 2008 and Object Oriented Programming

Internet and Web Programming

Client-Side Internet and Web Programming
Rich Internet Application (RIA) Development
Server-Side Internet and Web Programming

Fig. 2 | Deitel *Dive Into*® Series programming training courses. (Part 2 of 2.)

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, Mark Taub, Editor-in-Chief of Prentice Hall Professional, and John Fuller, Managing Editor of Prentice Hall Professional. Carole Snyder, Lisa Bailey and Dolores Mars did a remarkable job recruiting the book's large review team and managing the review process. Sandra Schroeder designed the book's cover. Scott Disanno and Robert Engelhardt managed the book's production.

This book was adapted from our book *C++ How to Program, 6/e*. We wish to acknowledge the efforts of our reviewers on that book. Adhering to a tight time schedule, they scru-

tinized the text and the programs, providing countless suggestions for improving the accuracy and completeness of the presentation.

C++ How to Program, 6/e *Reviewers*
Industry and Academic 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). **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.).

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.

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/cppfp/
```

We hope you enjoy reading *C++ for Programmers* as much as we enjoyed writing it!

Paul J. Deitel
Dr. Harvey M. Deitel

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. Through Deitel & Associates, Inc., he has delivered C++, Java, C, C# and Visual Basic courses to industry, government and military clients, including Cisco, IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA at the Kennedy Space Center, White Sands Missile Range, the National Severe Storm Laboratory, Rogue Wave Software, Boeing, Stratus, Hyperion Software, Adra Systems, Entergy, CableData Systems, Nortel Networks, Puma, iRobot, Invensys and many more. He has lectured on C++ and Java for the Boston Chapter of the Association for Computing Machinery, and on .NET technologies for ITESM in Monterrey, Mexico. 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 47 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. The Deitels' texts have earned international recognition with translations published in Japanese, German, Russian, Spanish, Traditional Chinese, Simplified Chinese, Korean, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish. 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 web software technology, object technology education and Internet business development through its Internet Business Initiative. The company provides instructor-led professional courses on major programming languages and platforms, such as C++, Java, C, C#, Visual C++, Visual Basic, XML, Perl, Python, object technology and Internet and web programming. The founders of Deitel & Associates, Inc., are Paul J. Deitel and Dr. Harvey M. Deitel. The company's clients include many of the world's largest companies, government agencies, branches of the military, and academic institutions. Through its 32-year publishing partnership with Prentice Hall, Deitel & Associates, Inc. publishes leading-edge programming professional books, textbooks, *LiveLessons* video courses, interactive multimedia *Cyber Classrooms*, web-based training courses and e-content for popular course management systems. 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 *Dive-Into*® Series Corporate Training curriculum offered on-site at clients worldwide, 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:

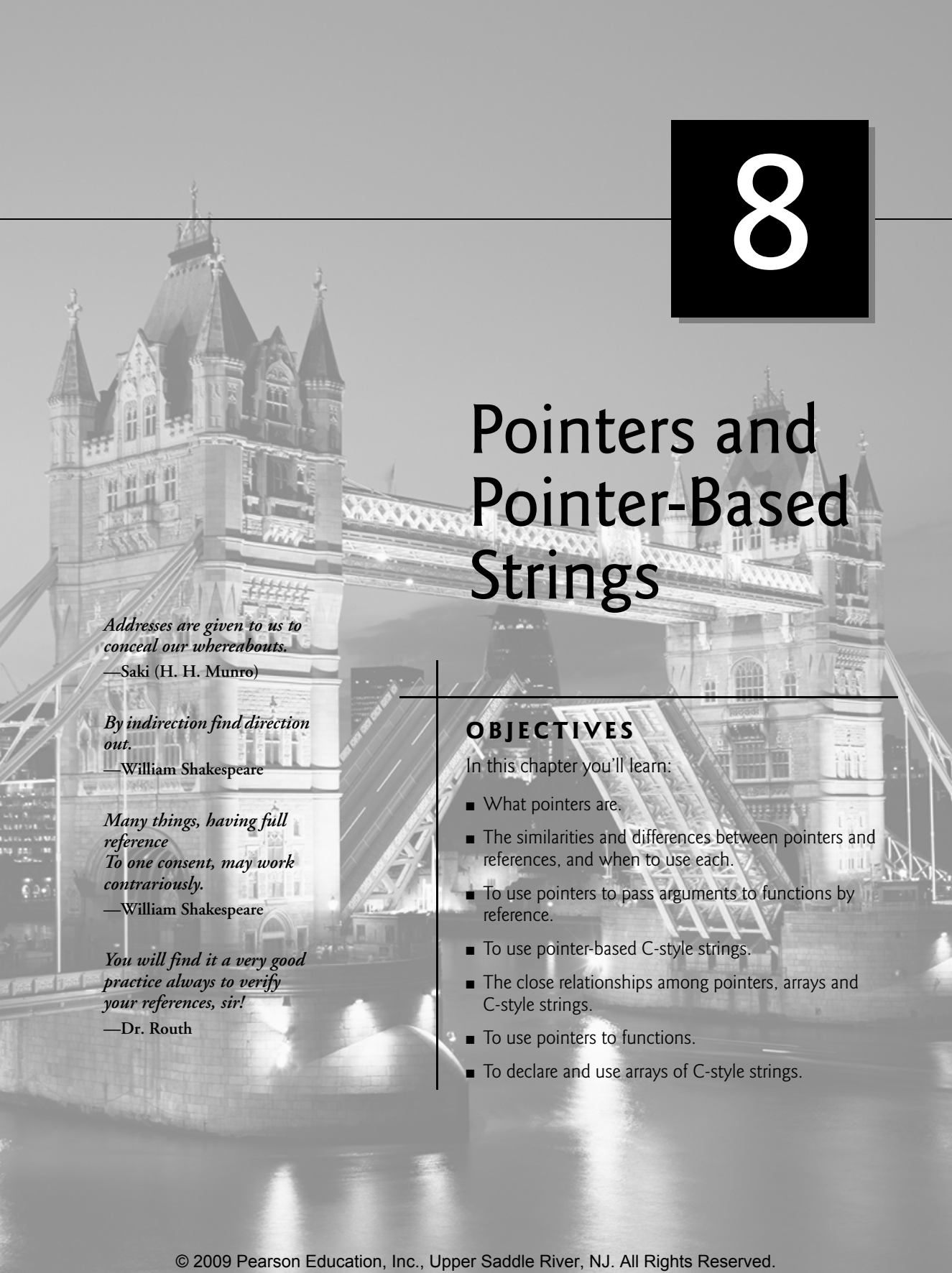
```
www.deitel.com/resourcecenters.html
```

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
```



8

Pointers and Pointer-Based Strings

Addresses are given to us to conceal our whereabouts.
—Saki (H. H. Munro)

By indirection find direction out.
—William Shakespeare

Many things, having full reference
To one consent, may work contrariously.
—William Shakespeare

You will find it a very good practice always to verify your references, sir!
—Dr. Routh

OBJECTIVES

In this chapter you'll learn:

- What pointers are.
- The similarities and differences between pointers and references, and when to use each.
- To use pointers to pass arguments to functions by reference.
- To use pointer-based C-style strings.
- The close relationships among pointers, arrays and C-style strings.
- To use pointers to functions.
- To declare and use arrays of C-style strings.

Outline

8.1 Introduction

This chapter discusses one of the most powerful features of the C++ programming language, the pointer. In Chapter 6, we saw that references can be used to perform pass-by-reference. Pointers also enable pass-by-reference and can be used to create and manipulate dynamic data structures (i.e., data structures that can grow and shrink), such as linked lists, queues, stacks and trees. This chapter explains basic pointer concepts and reinforces the intimate relationship among arrays and pointers. The view of arrays as pointers derives from the C programming language. As we saw in Chapter 7, C++ Standard Library class `vector` provides an implementation of arrays as full-fledged objects.

Similarly, C++ actually offers two types of strings—`string` class objects (which we have been using since Chapter 3) and C-style, `char *` pointer-based strings. This chapter on pointers discusses `char *` strings to deepen your knowledge of pointers. In fact, the null-terminated strings that we introduced in Section 7.4 and used in Fig. 7.12 are `char *` pointer-based strings. C-style, `char *` pointer-based strings are widely used in legacy C and C++ systems. So, if you work with legacy C or C++ systems, you may be required to manipulate these `char *` pointer-based strings.

We'll examine the use of pointers with classes in Chapter 13, Object-Oriented Programming: Polymorphism, where we'll see that the so-called "polymorphic processing" of object-oriented programming is performed with pointers and references.

8.2 Pointer Variable Declarations and Initialization

Pointer variables contain memory addresses as their values. Normally, a variable directly contains a specific value. However, a pointer contains the memory address of a variable that, in turn, contains a specific value. In this sense, a variable name ***directly references a value***, and a pointer ***indirectly references a value*** (Fig. 8.1). Referencing a value through a pointer

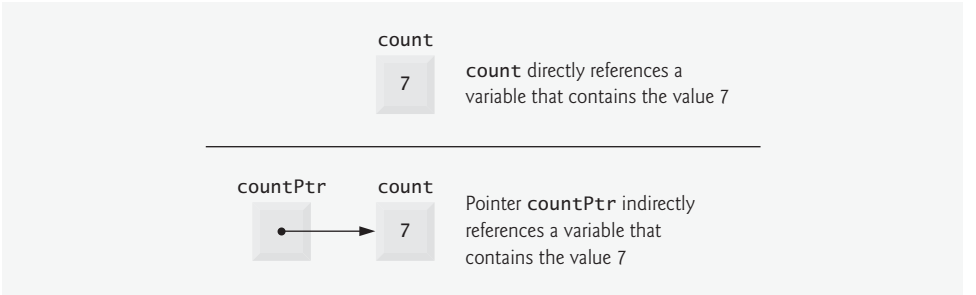


Fig. 8.1 | Directly and indirectly referencing a variable.

is often called ***indirection***. Note that diagrams typically represent a pointer as an arrow from the variable that contains an address to the variable located at that address in memory.

Pointers, like any other variables, must be declared before they can be used. For example, for the pointer in Fig. 8.1, the declaration

```
int *countPtr, count;
```

declares the variable `countPtr` to be of type `int *` (i.e., a pointer to an `int` value) and is read, "`countPtr` is a pointer to `int`" or "`countPtr` points to an object of type `int`." Also, variable `count` in the preceding declaration is declared to be an `int`, not a pointer to an `int`. The `*` in the declaration applies only to `countPtr`. Each variable being declared as a pointer must be preceded by an asterisk (`*`). For example, the declaration

```
double *xPtr, *yPtr;
```

indicates that both `xPtr` and `yPtr` are pointers to `double` values. When `*` appears in a declaration, it is not an operator; rather, it indicates that the variable being declared is a pointer. Pointers can be declared to point to objects of any data type.

Common Programming Error 8.1

*Assuming that the * used to declare a pointer distributes to all variable names in a declaration's comma-separated list of variables can lead to errors. Each pointer must be declared with the * prefixed to the name (either with or without a space in between—the compiler ignores the space). Declaring only one variable per declaration helps avoid these types of errors and improves program readability.*

Good Programming Practice 8.1

Although it is not a requirement, including the letters `Ptr` in pointer variable names makes it clear that these variables are pointers and that they must be handled accordingly.

Pointers should be initialized either when they are declared or in an assignment. A pointer may be initialized to `0`, `NULL` or an address of the corresponding type. A pointer with the value `0` or `NULL` points to nothing and is known as a ***null pointer***. Symbolic constant `NULL` is defined in header file `<iostream>` (and in several other standard library header files) to represent the value `0`. Initializing a pointer to `NULL` is equivalent to initializing a pointer to `0`, but in C++, `0` is used by convention. When `0` is assigned, it is converted to a pointer of the appropriate type. The value `0` is the only integer value that can be

assigned directly to a pointer variable without first casting the integer to a pointer type. Assigning a variable's numeric address to a pointer is discussed in Section 8.3.

Error-Prevention Tip 8.1

Initialize pointers to prevent pointing to unknown or uninitialized areas of memory.

8.3 Pointer Operators

The ***address operator (&)*** is a unary operator that obtains the memory address of its operand. For example, assuming the declarations

```
int y = 5; // declare variable y
int *yPtr; // declare pointer variable yPtr
```

the statement

```
yPtr = &y; // assign address of y to yPtr
```

assigns the address of the variable `y` to pointer variable `yPtr`. Then variable `yPtr` is said to "point to" `y`. Now, `yPtr` indirectly references variable y's value. Note that the use of the `&` in the preceding statement is not the same as the use of the `&` in a reference variable declaration, which is always preceded by a data-type name. When declaring a reference, the `&` is part of the type. In an expression like `&y`, the `&` is an operator.

Figure 8.2 shows a schematic representation of memory after the preceding assignment. The "pointing relationship" is indicated by drawing an arrow from the box that represents the pointer `yPtr` in memory to the box that represents the variable `y` in memory.

Figure 8.3 shows another representation of the pointer in memory, assuming that integer variable `y` is stored at memory location `600000` and that pointer variable `yPtr` is stored at memory location `500000`. The operand of the address operator must be an *lvalue* (i.e., something to which a value can be assigned, such as a variable name or a reference); the address operator cannot be applied to constants or to expressions that do not result in references.

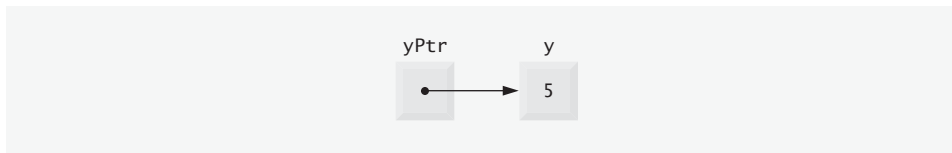


Fig. 8.2 | Graphical representation of a pointer pointing to a variable in memory.

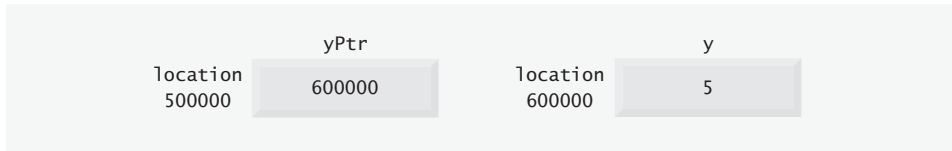


Fig. 8.3 | Representation of `y` and `yPtr` in memory.

The * ***operator***, commonly referred to as the ***indirection operator*** or ***dereferencing operator***, returns a synonym (i.e., an alias or a nickname) for the object to which its pointer operand points. For example (referring again to Fig. 8.2), the statement

```
cout << *yPtr << endl;
```

prints the value of variable y, namely, 5, just as the statement

```
cout << y << endl;
```

would. Using * in this manner is called ***dereferencing a pointer***. Note that a dereferenced pointer may also be used on the left side of an assignment statement, as in

```
*yPtr = 9;
```

which would assign 9 to y in Fig. 8.3. The dereferenced pointer may also be used to receive an input value as in

```
cin >> *yPtr;
```

which places the input value in y. The dereferenced pointer is an *lvalue*.

Common Programming Error 8.2

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion, possibly with incorrect results.

Common Programming Error 8.3

An attempt to dereference a variable that is not a pointer is a compilation error.

Common Programming Error 8.4

Dereferencing a null pointer is often a fatal execution-time error.

The program in Fig. 8.4 demonstrates the & and * pointer operators. Memory locations are output by << in this example as hexadecimal (i.e., base-16) integers. Note that the hexadecimal memory addresses output by this program are compiler and operating-system dependent, so you may get different results when you run the program.

Portability Tip 8.1

The format in which a pointer is output is compiler dependent. Some output pointer values as hexadecimal integers, some use decimal integers and some use other formats.

Notice that the address of a (line 15) and the value of aPtr (line 16) are identical in the output, confirming that the address of a is indeed assigned to the pointer variable aPtr. The & and * operators are inverses of one another—when they are both applied consecutively to aPtr in either order, they "cancel one another out" and the same result (the value in aPtr) is printed.

```
1  // Fig. 8.4: fig08_04.cpp
2  // Pointer operators & and *.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int a; // a is an integer
10    int *aPtr; // aPtr is an int * -- pointer to an integer
11
12    a = 7; // assigned 7 to a
13    aPtr = &a; // assign the address of a to aPtr
14
15    cout << "The address of a is " << &a
16       << "\nThe value of aPtr is " << aPtr;
17    cout << "\n\nThe value of a is " << a
18       << "\nThe value of *aPtr is " << *aPtr;
19    cout << "\n\nShowing that * and & are inverses of "
20       << "each other.\n&*aPtr = " << &*aPtr
21       << "\n*&aPtr = " << *&aPtr << endl;
22    return 0; // indicates successful termination
23 } // end main
```

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

Fig. 8.4 | Pointer operators & and *.

Figure 8.5 lists the precedence and associativity of the operators introduced to this point. Note that the address operator (&) and the dereferencing operator (*) are unary operators on the third level of precedence in the chart.

Operators	Associativity	Type
`()` `[]`	left to right	highest
`++` `--` ***`static_cast`***`<` *type* `>(` *operand* `)`	left to right	unary (postfix)
`++` `--` `+` `-` `!` `&` `*`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive

Fig. 8.5 | Operator precedence and associativity. (Part 1 of 2.)

Operators	Associativity	Type
<< >>	left to right	insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 8.5 | Operator precedence and associativity. (Part 2 of 2.)

8.4 Passing Arguments to Functions by Reference with Pointers

There are three ways in C++ to pass arguments to a function—pass-by-value, ***pass-by-reference with reference arguments*** and ***pass-by-reference with pointer arguments***. Chapter 6 compared and contrasted pass-by-value and pass-by-reference with reference arguments. In this section, we explain pass-by-reference with pointer arguments.

As we saw in Chapter 6, `return` can be used to return one value from a called function to a caller (or to return control from a called function without passing back a value). We also saw that arguments can be passed to a function using reference arguments. Such arguments enable the called function to modify the original values of the arguments in the caller. Reference arguments also enable programs to pass large data objects to a function and avoid the overhead of passing the objects by value (which, of course, requires making a copy of the object). Pointers, like references, also can be used to modify one or more variables in the caller or to pass pointers to large data objects to avoid the overhead of passing the objects by value.

In C++, programmers can use pointers and the indirection operator (*) to accomplish pass-by-reference (exactly as pass-by-reference is done in C programs—C does not have references). When calling a function with an argument that should be modified, the address of the argument is passed. This is normally accomplished by applying the address operator (`&`) to the name of the variable whose value will be modified.

As we saw in Chapter 7, arrays are not passed using operator `&`, because the name of the array is the starting location in memory of the array (i.e., an array name is already a pointer). The name of an array, `arrayName`, is equivalent to `&arrayName[ 0 ]`. When the address of a variable is passed to a function, the indirection operator (*) can be used in the function to form a synonym for the name of the variable—this in turn can be used to modify the value of the variable at that location in the caller's memory.

Figure 8.6 and Fig. 8.7 present two versions of a function that cubes an integer—`cubeByValue` and `cubeByReference`. Figure 8.6 passes variable `number` by value to function `cubeByValue` (line 15). Function `cubeByValue` (lines 21–24) cubes its argument and passes the new value back to `main` using a `return` statement (line 23). The new value is

```
1  // Fig. 8.6: fig08_06.cpp
2  // Pass-by-value used to cube a variable's value.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int cubeByValue( int ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    number = cubeByValue( number ); // pass number by value to cubeByValue
16    cout << "\nThe new value of number is " << number << endl;
17    return 0; // indicates successful termination
18 } // end main
19
20 // calculate and return cube of integer argument
21 int cubeByValue( int n )
22 {
23    return n * n * n; // cube local variable n and return result
24 } // end function cubeByValue
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 8.6 | Pass-by-value used to cube a variable's value.

assigned to `number` (line 15) in `main`. Note that the calling function has the opportunity to examine the result of the function call before modifying variable `number`'s value. For example, in this program, we could have stored the result of `cubeByValue` in another variable, examined its value and assigned the result to `number` only after determining that the returned value was reasonable.

Figure 8.7 passes the variable `number` to function `cubeByReference` using pass-by-reference with a pointer argument (line 16)—the address of `number` is passed to the function. Function `cubeByReference` (lines 23–26) specifies parameter `nPtr` (a pointer to `int`) to receive its argument. The function dereferences the pointer and cubes the value to which `nPtr` points (line 25). This directly changes the value of `number` in `main`.

Common Programming Error 8.5

Not dereferencing a pointer when it is necessary to do so to obtain the value to which the pointer points is an error.

A function receiving an address as an argument must define a pointer parameter to receive the address. For example, the header for function `cubeByReference` (line 23) specifies that `cubeByReference` receives the address of an `int` variable (i.e., a pointer to an `int`) as an argument, stores the address locally in `nPtr` and does not return a value.

The function prototype for `cubeByReference` (line 8) contains `int *` in parentheses. As with other variable types, it is not necessary to include names of pointer parameters in

```
1  // Fig. 8.7: fig08_07.cpp
2  // Pass-by-reference with a pointer argument used to cube a
3  // variable's value.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  void cubeByReference( int * ); // prototype
9
10 int main()
11 {
12    int number = 5;
13
14    cout << "The original value of number is " << number;
15
16    cubeByReference( &number ); // pass number address to cubeByReference
17
18    cout << "\nThe new value of number is " << number << endl;
19    return 0; // indicates successful termination
20 } // end main
21
22 // calculate cube of *nPtr; modifies variable number in main
23 void cubeByReference( int *nPtr )
24 {
25    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
26 } // end function cubeByReference
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 8.7 | Pass-by-reference with a pointer argument used to cube a variable's value.

function prototypes. Parameter names included for documentation purposes are ignored by the compiler.

Figures 8.8–8.9 analyze graphically the execution of the programs in Fig. 8.6 and Fig. 8.7, respectively.

Software Engineering Observation 8.1

Use pass-by-value to pass arguments to a function unless the caller explicitly requires that the called function directly modify the value of the argument variable in the caller. This is another example of the principle of least privilege.

In the function header and in the prototype for a function that expects a one-dimensional array as an argument, the pointer notation in the parameter list of `cubeByReference` may be used. The compiler does not differentiate between a function that receives a pointer and a function that receives a one-dimensional array. This, of course, means that the function must "know" when it is receiving an array or simply a single variable which is being passed by reference. When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b` (pronounced "`b` is a pointer to an integer"). Both forms of declaring a function parameter as a one-dimensional array are interchangeable.

Step 1: Before `main` calls `cubeByValue`:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

Step 2: After `cubeByValue` receives the call:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   125
   return n * n * n;
}
```

n: 5

Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:

```
int main()
{
   int number = 5;
       125
   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

Step 5: After `main` completes the assignment to `number`:

```
int main()
{
   int number = 5;
    125          125
   number = cubeByValue( number );
}
```

number: 125

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

Fig. 8.8 | Pass-by-value analysis of the program of Fig. 8.6.

Fig. 8.9 | Pass-by-reference analysis (with a pointer argument) of the program of Fig. 8.7.

8.5 Using const with Pointers

Recall that the const qualifier enables you to inform the compiler that the value of a particular variable should not be modified.

Over the years, a large base of legacy code was written in early versions of C that did not use const, because it was not available. For this reason, there are great opportunities for improvement in the software engineering of old (also called "legacy") C code. Also, many programmers currently using ANSI C and C++ do not use const in their programs, because they began programming in early versions of C. These programmers are missing many opportunities for good software engineering.

Many possibilities exist for using (or not using) const with function parameters. How do you choose the most appropriate of these possibilities? Let the principle of least privilege be your guide. Always award a function enough access to the data in its parameters to accomplish its specified task, but no more. This section discusses how to combine const with pointer declarations to enforce the principle of least privilege.

Chapter 6 explained that when a function is called using pass-by-value, a copy of the argument (or arguments) in the function call is made and passed to the function. If the copy

is modified in the function, the original value is maintained in the caller without change. In many cases, a value passed to a function is modified so that the function can accomplish its task. However, in some instances, the value should not be altered in the called function, even though the called function manipulates only a copy of the original value.

For example, consider a function that takes a one-dimensional array and its size as arguments and subsequently prints the array. Such a function should loop through the array and output each array element individually. The size of the array is used in the function body to determine the highest subscript of the array so the loop can terminate when the printing completes. The size of the array does not change in the function body, so it should be declared `const`. Of course, because the array is only being printed, it, too, should be declared `const`. This is especially important because an entire array is *always* passed by reference and could easily be changed in the called function.

Software Engineering Observation 8.2

If a value does not (or should not) change in the body of a function to which it is passed, the parameter should be declared `const` to ensure that it is not accidentally modified.

If an attempt is made to modify a `const` value, a warning or an error is issued, depending on the particular compiler.

Error-Prevention Tip 8.2

Before using a function, check its function prototype to determine the parameters that it can modify.

There are four ways to pass a pointer to a function: a nonconstant pointer to nonconstant data (Fig. 8.10), a nonconstant pointer to constant data (Fig. 8.11 and Fig. 8.12), a constant pointer to nonconstant data (Fig. 8.13) and a constant pointer to constant data (Fig. 8.14). Each combination provides a different level of access privileges.

Nonconstant Pointer to Nonconstant Data

The highest access is granted by a ***nonconstant pointer to nonconstant data***—the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data. The declaration for such a pointer does not include `const`. Such a pointer can be used to receive a null-terminated string in a function that changes the pointer value to process (and possibly modify) each character in the string. Recall from Section 7.4 that a null-terminated string can be placed in a character array that contains the characters of the string and a null character indicating where the string ends.

In Fig. 8.10, function `convertToUppercase` (lines 25–34) declares parameter `sPtr` (line 25) to be a nonconstant pointer to nonconstant data (again, `const` is not used). The function processes one character at a time from the null-terminated string stored in character array `phrase` (lines 27–33). Keep in mind that a character array's name is really equivalent to a `const` pointer to the first character of the array, so passing `phrase` as an argument to `convertToUppercase` is possible. Function `islower` (line 29) takes a character argument and returns true if the character is a lowercase letter and false otherwise. Characters in the range `'a'` through `'z'` are converted to their corresponding uppercase letters by function `toupper` (line 30); others remain unchanged—function `toupper` takes

```
1  // Fig. 8.10: fig08_10.cpp
2  // Converting lowercase letters to uppercase letters
3  // using a nonconstant pointer to nonconstant data.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <cctype> // prototypes for islower and toupper
9  using std::islower;
10 using std::toupper;
11
12 void convertToUppercase( char * );
13
14 int main()
15 {
16    char phrase[] = "characters and $32.98";
17
18    cout << "The phrase before conversion is: " << phrase;
19    convertToUppercase( phrase );
20    cout << "\nThe phrase after conversion is:  " << phrase << endl;
21    return 0; // indicates successful termination
22 } // end main
23
24 // convert string to uppercase letters
25 void convertToUppercase( char *sPtr )
26 {
27    while ( *sPtr != '\0' ) // loop while current character is not '\0'
28    {
29       if ( islower( *sPtr ) ) // if character is lowercase,
30          *sPtr = toupper( *sPtr ); // convert to uppercase
31
32       sPtr++; // move sPtr to next character in string
33    } // end while
34 } // end function convertToUppercase
```

```
The phrase before conversion is: characters and $32.98
The phrase after conversion is:  CHARACTERS AND $32.98
```

Fig. 8.10 | Converting a string to uppercase letters using a nonconstant pointer to nonconstant data.

one character as an argument. If the character is a lowercase letter, the corresponding uppercase letter is returned; otherwise, the original character is returned. Function `toupper` and function `islower` are part of the character-handling library `<cctype>` (see Chapter 19, Bits, Characters, C-Strings and `structs`). After processing one character, line 32 increments `sPtr` by 1 (this would not be possible if `sPtr` were declared `const`). When operator `++` is applied to a pointer that points to an array, the memory address stored in the pointer is modified to point to the next element of the array (in this case, the next character in the string). Adding one to a pointer is one valid operation in ***pointer arithmetic***, which is covered in detail in Sections 8.8–8.9.

Nonconstant Pointer to Constant Data

A *nonconstant pointer to constant data* is a pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer. Such a pointer might be used to receive an array argument to a function that will process each element of the array, but should not be allowed to modify the data. For example, function `printCharacters` (lines 22–26 of Fig. 8.11) declares parameter `sPtr` (line 22) to be of type `const char *`, so that it can receive a null-terminated pointer-based string. The declaration is read from right to left as "`sPtr` is a pointer to a character constant." The body of the function uses a `for` statement (lines 24–25) to output each character in the string until the null character is encountered. After each character is printed, pointer `sPtr` is incremented to point to the next character in the string (this works because the pointer is not `const`). Function `main` creates `char` array `phrase` to be passed to `printCharacters`. Again, we can pass the array `phrase` to `printCharacters` because the name of the array is really a pointer to the first character in the array.

Figure 8.12 demonstrates the compilation error messages produced when attempting to compile a function that receives a nonconstant pointer to constant data, then tries to use that pointer to modify the data. [*Note:* Recall that compiler error messages may vary among compilers.]

```
1  // Fig. 8.11: fig08_11.cpp
2  // Printing a string one character at a time using
3  // a nonconstant pointer to constant data.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  void printCharacters( const char * ); // print using pointer to const data
9
10 int main()
11 {
12    const char phrase[] = "print characters of a string";
13
14    cout << "The string is:\n";
15    printCharacters( phrase ); // print characters in phrase
16    cout << endl;
17    return 0; // indicates successful termination
18 } // end main
19
20 // sPtr can be modified, but it cannot modify the character to which
21 // it points, i.e., sPtr is a "read-only" pointer
22 void printCharacters( const char *sPtr )
23 {
24    for ( ; *sPtr != '\0'; sPtr++ ) // no initialization
25       cout << *sPtr; // display character without modification
26 } // end function printCharacters
```

```
The string is:
print characters of a string
```

Fig. 8.11 | Printing a string one character at a time using a nonconstant pointer to constant data.

```
1  // Fig. 8.12: fig08_12.cpp
2  // Attempting to modify data through a
3  // nonconstant pointer to constant data.
4
5  void f( const int * ); // prototype
6
7  int main()
8  {
9     int y;
10
11    f( &y ); // f attempts illegal modification
12    return 0; // indicates successful termination
13 } // end main
14
15 // xPtr cannot modify the value of constant variable to which it points
16 void f( const int *xPtr )
17 {
18    *xPtr = 100; // error: cannot modify a const object
19 } // end function f
```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_12.cpp 18:
   Cannot modify a const object in function f(const int *)
```

Microsoft Visual C++ compiler error message:

```
c:\cppfp_examples\ch08\Fig08_12\fig08_12.cpp(18) :
   error C3892: 'xPtr' : you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig08_12.cpp: In function `void f(const int*)':
fig08_12.cpp:18: error: assignment of read-only location
```

Fig. 8.12 | Attempting to modify data through a nonconstant pointer to constant data.

As we know, arrays are aggregate data types that store related data items of the same type under one name. When a function is called with an array as an argument, the array is passed to the function by reference. However, objects are always passed by value—a copy of the entire object is passed. This requires the execution-time overhead of making a copy of each data item in the object and storing it on the function call stack. When an object must be passed to a function, we can use a pointer to constant data (or a reference to constant data) to get the performance of pass-by-reference and the protection of pass-by-value. When a pointer to an object is passed, only a copy of the address of the object must be made—the object itself is not copied. On a machine with four-byte addresses, a copy of four bytes of memory is made rather than a copy of a possibly large object.

Performance Tip 8.1

If they do not need to be modified by the called function, pass large objects using pointers to constant data or references to constant data, to obtain the performance benefits of pass-by-reference.

Software Engineering Observation 8.3

Pass large objects using pointers to constant data, or references to constant data, to obtain the security of pass-by-value.

Constant Pointer to Nonconstant Data

A ***constant pointer to nonconstant data*** is a pointer that always points to the same memory location; the data at that location can be modified through the pointer. An example of such a pointer is an array name, which is a constant pointer to the beginning of the array. All data in the array can be accessed and changed by using the array name and array subscripting. A constant pointer to nonconstant data can be used to receive an array as an argument to a function that accesses array elements using array subscript notation. Pointers that are declared `const` must be initialized when they are declared. (If the pointer is a function parameter, it is initialized with a pointer that is passed to the function.) The program of Fig. 8.13 attempts to modify a constant pointer. Line 11 declares pointer `ptr` to be of type `int * const`. The declaration in the figure is read from right to left as "`ptr` is a constant pointer to a nonconstant integer." The pointer is initialized with the address of integer variable `x`. Line 14 attempts to assign the address of `y` to `ptr`, but the compiler generates an error message. Note that no error occurs when line 13 assigns the value `7` to

```
1  // Fig. 8.13: fig08_13.cpp
2  // Attempting to modify a constant pointer to nonconstant data.
3
4  int main()
5  {
6     int x, y;
7
8     // ptr is a constant pointer to an integer that can
9     // be modified through ptr, but ptr always points to the
10    // same memory location.
11    int * const ptr = &x; // const pointer must be initialized
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign to it a new address
15    return 0; // indicates successful termination
16 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_13.cpp 14: Cannot modify a const object in function main()
```

Microsoft Visual C++ compiler error message:

```
c:\cppfp_examples\ch08\Fig08_13\fig08_13.cpp(14) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig08_13.cpp: In function `int main()':
fig08_13.cpp:14: error: assignment of read-only variable `ptr'
```

Fig. 8.13 | Attempting to modify a constant pointer to nonconstant data.

*ptr—the nonconstant value to which ptr points can be modified using the dereferenced ptr, even though ptr itself has been declared const.

Common Programming Error 8.6

Not initializing a pointer that is declared const is a compilation error.

Constant Pointer to Constant Data

The least amount of access privilege is granted by a ***constant pointer to constant data***. Such a pointer always points to the same memory location, and the data at that memory location cannot be modified using the pointer. This is how an array should be passed to a function that only reads the array, using array subscript notation, and does not modify the array. The program of Fig. 8.14 declares pointer variable ptr to be of type const int * const (line 14). This declaration is read from right to left as "ptr is a constant pointer to an integer constant." The figure shows the error messages generated when an attempt is

```
1  // Fig. 8.14: fig08_14.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 5, y;
10
11    // ptr is a constant pointer to a constant integer.
12    // ptr always points to the same location; the integer
13    // at that location cannot be modified.
14    const int *const ptr = &x;
15
16    cout << *ptr << endl;
17
18    *ptr = 7; // error: *ptr is const; cannot assign new value
19    ptr = &y; // error: ptr is const; cannot assign new address
20    return 0; // indicates successful termination
21 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_14.cpp 18: Cannot modify a const object in function main()
Error E2024 fig08_14.cpp 19: Cannot modify a const object in function main()
```

Microsoft Visual C++ compiler error message:

```
c:\cppfp_examples\ch08\Fig08_14\fig08_14.cpp(18) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
c:\cppfp_examples\ch08\Fig08_14\fig08_14.cpp(19) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
```

Fig. 8.14 | Attempting to modify a constant pointer to constant data. (Part 1 of 2.)

GNU C++ compiler error message:

```
fig08_14.cpp: In function `int main()':
fig08_14.cpp:18: error: assignment of read-only location
fig08_14.cpp:19: error: assignment of read-only variable `ptr'
```

Fig. 8.14 | Attempting to modify a constant pointer to constant data. (Part 2 of 2.)

made to modify the data to which `ptr` points (line 18) and when an attempt is made to modify the address stored in the pointer variable (line 19). Note that no errors occur when the program attempts to dereference `ptr`, or when the program attempts to output the value to which `ptr` points (line 16), because neither the pointer nor the data it points to is being modified in this statement.

8.6 Selection Sort Using Pass-by-Reference

In this section, we define a sorting program to demonstrate passing arrays and individual array elements by reference. We use the ***selection sort*** algorithm, which is an easy-to-program, but unfortunately inefficient, sorting algorithm. The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element. The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index. After the i^{th} iteration, the smallest i items of the array will be sorted into increasing order in the first i elements of the array.

As an example, consider the array

```
34   56   4   10   77   51   93   30   5   52
```

A program that implements the selection sort first determines the smallest value (4) in the array, which is contained in element 2. The program swaps the 4 with the value in element 0 (34), resulting in

```
4   56   34   10   77   51   93   30   5   52
```

[*Note:* We use bold to highlight the values that were swapped.] The program then determines the smallest value of the remaining elements (all elements except 4), which is 5, contained in element 8. The program swaps the 5 with the 56 in element 1, resulting in

```
4   5   34   10   77   51   93   30   56   52
```

On the third iteration, the program determines the next smallest value, 10, and swaps it with the value in element 2 (34).

```
4   5   10   34   77   51   93   30   56   52
```

The process continues until the array is fully sorted.

```
4   5   10   30   34   51   52   56   77   93
```

Note that after the first iteration, the smallest element is in the first position. After the second iteration, the two smallest elements are in order in the first two positions. After the third iteration, the three smallest elements are in order in the first three positions.

Figure 8.15 implements selection sort using two functions—`selectionSort` and `swap`. Function `selectionSort` (lines 36–53) sorts the array. Line 38 declares the variable `smallest`, which will store the index of the smallest element in the remaining array. Lines 41–52 loop `size - 1` times. Line 43 sets the index of the smallest element to the current index. Lines 46–49 loop over the remaining elements in the array. For each of these elements, line 48 compares its value to the value of the smallest element. If the current element is smaller than the smallest element, line 49 assigns the current element's index to `smallest`. When this loop finishes, `smallest` will contain the index of the smallest element in the remaining array. Line 51 calls function `swap` (lines 57–62) to place the smallest remaining element in the next spot in the array (i.e., exchange the array elements `array[i]` and `array[smallest]`).

Let us now look more closely at function `swap`. Remember that C++ enforces information hiding between functions, so `swap` does not have access to individual array elements in `selectionSort`. Because `selectionSort` *wants* `swap` to have access to the array elements to be swapped, `selectionSort` passes each of these elements to `swap` by refer-

```
1  // Fig. 8.15: fig08_15.cpp
2  // Selection sort with pass-by-reference. This program puts values into an
3  // array, sorts them into ascending order and prints the resulting array.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 void selectionSort( int * const, const int ); // prototype
12 void swap( int * const, int * const ); // prototype
13
14 int main()
15 {
16    const int arraySize = 10;
17    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19    cout << "Data items in original order\n";
20
21    for ( int i = 0; i < arraySize; i++ )
22       cout << setw( 4 ) << a[ i ];
23
24    selectionSort( a, arraySize ); // sort the array
25
26    cout << "\nData items in ascending order\n";
27
28    for ( int j = 0; j < arraySize; j++ )
29       cout << setw( 4 ) << a[ j ];
30
31    cout << endl;
32    return 0; // indicates successful termination
33 } // end main
```

Fig. 8.15 | Selection sort with pass-by-reference. (Part 1 of 2.)

```
34
35 // function to sort an array
36 void selectionSort( int * const array, const int size )
37 {
38    int smallest; // index of smallest element
39
40    // loop over size - 1 elements
41    for ( int i = 0; i < size - 1; i++ )
42    {
43       smallest = i; // first index of remaining array
44
45       // loop to find index of smallest element
46       for ( int index = i + 1; index < size; index++ )
47
48          if ( array[ index ] < array[ smallest ] )
49             smallest = index;
50
51       swap( &array[ i ], &array[ smallest ] );
52    } // end if
53 } // end function selectionSort
54
55 // swap values at memory locations to which
56 // element1Ptr and element2Ptr point
57 void swap( int * const element1Ptr, int * const element2Ptr )
58 {
59    int hold = *element1Ptr;
60    *element1Ptr = *element2Ptr;
61    *element2Ptr = hold;
62 } // end function swap
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 8.15 | Selection sort with pass-by-reference. (Part 2 of 2.)

ence—the address of each array element is passed explicitly. Although entire arrays are passed by reference, individual array elements are scalars and are ordinarily passed by value. Therefore, `selectionSort` uses the address operator (`&`) on each array element in the `swap` call (line 51) to effect pass-by-reference. Function `swap` (lines 57–62) receives `&array[ i ]` in pointer variable `element1Ptr`. Information hiding prevents `swap` from "knowing" the name `array[ i ]`, but `swap` can use `*element1Ptr` as a synonym for `array[ i ]`. Thus, when `swap` references `*element1Ptr`, it is actually referencing `array[ i ]` in `selectionSort`. Similarly, when `swap` references `*element2Ptr`, it is actually referencing `array[ smallest ]` in `selectionSort`.

Even though `swap` is not allowed to use the statements

```
hold = array[ i ];
array[ i ] = array[ smallest ];
array[ smallest ] = hold;
```

precisely the same effect is achieved by

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

in the `swap` function of Fig. 8.15.

Several features of function `selectionSort` should be noted. The function header (line 36) declares `array` as `int * const array`, rather than `int array[]`, to indicate that the function receives a one-dimensional array as an argument. Both parameter `array`'s pointer and parameter `size` are declared `const` to enforce the principle of least privilege. Although parameter `size` receives a copy of a value in `main` and modifying the copy cannot change the value in `main`, `selectionSort` does not need to alter `size` to accomplish its task—the array size remains fixed during the execution of `selectionSort`. Therefore, `size` is declared `const` to ensure that it is not modified. If the size of the array were to be modified during the sorting process, the sorting algorithm would not run correctly.

Note that function `selectionSort` receives the size of the array as a parameter, because the function must have that information to sort the array. When an array is passed to a function, only the memory address of the first element of the array is received by the function; the array size must be passed separately to the function.

By defining function `selectionSort` to receive the array size as a parameter, we enable the function to be used by any program that sorts one-dimensional `int` arrays of arbitrary size. The size of the array could have been programmed directly into the function, but this would restrict the function to processing an array of a specific size and reduce the function's reusability—only programs processing one-dimensional `int` arrays of the specific size "hard coded" into the function could use the function.

Software Engineering Observation 8.4

When passing an array to a function, also pass the size of the array (rather than building into the function knowledge of the array size)—this makes the function more reusable.

8.7 sizeof Operator

C++ provides the unary operator **`sizeof`** to determine the size of an array (or of any other data type, variable or constant) in bytes during program compilation. When applied to the name of an array, as in Fig. 8.16 (line 14), the `sizeof` operator returns the total number of bytes in the array as a value of type `size_t` (an unsigned integer type that is at least as big as `unsigned int`). Note that this is different from the `size` of a `vector< int >`, for example, which is the number of integer elements in the `vector`. The computer we used to compile this program stores variables of type `double` in 8 bytes of memory, and `array` is declared to have 20 elements (line 12), so `array` uses 160 bytes in memory. When applied to a pointer parameter (line 24) in a function that receives an array as an argument, the `sizeof` operator returns the size of the pointer in bytes (4 on the system we used)—not the size of the array.

Common Programming Error 8.7

Using the `sizeof` operator in a function to find the size in bytes of an array parameter results in the size in bytes of a pointer, not the size in bytes of the array.

```
1  // Fig. 8.16: fig08_16.cpp
2  // Sizeof operator when used on an array name
3  // returns the number of bytes in the array.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  size_t getSize( double * ); // prototype
9
10 int main()
11 {
12    double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system
13
14    cout << "The number of bytes in the array is " << sizeof( array );
15
16    cout << "\nThe number of bytes returned by getSize is "
17       << getSize( array ) << endl;
18    return 0; // indicates successful termination
19 } // end main
20
21 // return size of ptr
22 size_t getSize( double *ptr )
23 {
24    return sizeof( ptr );
25 } // end function getSize
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

Fig. 8.16 | `sizeof` operator when applied to an array name returns the number of bytes in the array.

[*Note:* When the Borland C++ compiler is used to compile Fig. 8.16, the compiler generates the warning message "`Parameter 'ptr' is never used in function getSize(double *)`." This warning occurs because `sizeof` is actually a compile-time operator; thus, variable `ptr` is not used in the function's body at execution time. Many compilers issue warnings like this to let you know that a variable is not being used so that you can either remove it from your code or modify your code to use the variable properly. Similar messages occur in Fig. 8.17 with various compilers.]

The number of elements in an array also can be determined using the results of two `sizeof` operations. For example, consider the following array declaration:

```
double realArray[ 22 ];
```

If variables of data type `double` are stored in eight bytes of memory, array `realArray` contains a total of 176 bytes. To determine the number of elements in the array, the following expression (which is evaluated at compile time) can be used:

```
sizeof realArray / sizeof( double ) // calculate number of elements
```

The expression determines the number of bytes in array `realArray` (176) and divides that value by the number of bytes used in memory to store a `double` value (8)—the result is the number of elements in `realArray` (22).

Determining the Sizes of the Fundamental Types, an Array and a Pointer
Figure 8.17 uses `sizeof` to calculate the number of bytes used to store most of the standard data types. Notice that, in the output, the types `double` and `long double` have the same size. Types may have different sizes based on the platform running the program. On another system, for example, `double` and `long double` may be of different sizes.

```
1  // Fig. 8.17: fig08_17.cpp
2  // Demonstrating the sizeof operator.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     char c; // variable of type char
10    short s; // variable of type short
11    int i; // variable of type int
12    long l; // variable of type long
13    float f; // variable of type float
14    double d; // variable of type double
15    long double ld; // variable of type long double
16    int array[ 20 ]; // array of int
17    int *ptr = array; // variable of type int *
18
19    cout << "sizeof c = " << sizeof c
20       << "\tsizeof(char) = " << sizeof( char )
21       << "\nsizeof s = " << sizeof s
22       << "\tsizeof(short) = " << sizeof( short )
23       << "\nsizeof i = " << sizeof i
24       << "\tsizeof(int) = " << sizeof( int )
25       << "\nsizeof l = " << sizeof l
26       << "\tsizeof(long) = " << sizeof( long )
27       << "\nsizeof f = " << sizeof f
28       << "\tsizeof(float) = " << sizeof( float )
29       << "\nsizeof d = " << sizeof d
30       << "\tsizeof(double) = " << sizeof( double )
31       << "\nsizeof ld = " << sizeof ld
32       << "\tsizeof(long double) = " << sizeof( long double )
33       << "\nsizeof array = " << sizeof array
34       << "\nsizeof ptr = " << sizeof ptr << endl;
35    return 0; // indicates successful termination
36 } // end main
```

```
sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 4    sizeof(long) = 4
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 8   sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

Fig. 8.17 | `sizeof` operator used to determine standard data type sizes.

Portability Tip 8.2

The number of bytes used to store a particular data type may vary among systems. When writing programs that depend on data type sizes, and that will run on several computer systems, use `sizeof` *to determine the number of bytes used to store the data types.*

Operator `sizeof` can be applied to any expression or type name. When `sizeof` is applied to a variable name (which is not an array name) or other expression, the number of bytes used to store the specific type of the expression's value is returned. Note that the parentheses used with `sizeof` are required only if a type name (e.g., `int`) is supplied as its operand. The parentheses used with `sizeof` are not required when `sizeof`'s operand is an expression. Remember that `sizeof` is an operator, not a function, and that it has its effect at compile time, not execution time.

Common Programming Error 8.8

Omitting the parentheses in a `sizeof` *operation when the operand is a type name is a compilation error.*

Performance Tip 8.2

Because `sizeof` *is a compile-time unary operator, not an execution-time operator, using* `sizeof` *does not negatively impact execution performance.*

Error-Prevention Tip 8.3

To avoid errors associated with omitting the parentheses around the operand of operator `sizeof`*, many programmers include parentheses around every* `sizeof` *operand.*

8.8 Pointer Expressions and Pointer Arithmetic

8.9 Relationship Between Pointers and Arrays

8.10 Arrays of Pointers

8.11 Case Study: Card Shuffling and Dealing Simulation

8.12 Function Pointers

8.13 Introduction to Pointer-Based String Processing

8.14 Wrap-Up