More than 450,000 Programmers
Have Learned C++ from Previous Editions

C++ PRIMER

FOURTH EDITION

Stanley B. Lippman
Josée Lajoie
Barbara E. Moo

Library of Congress Cataloging-in-Publication Data

Lippman, Stanley B.
C++ primer / Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. — 4th ed.
p. cm.
Includes index.
ISBN 0-201-72148-1 (pbk. : alk. paper)
1. C++ (Computer program language) I. Lajoie, Josée. II. Moo, Barbara E. III. Title.

QA76.73.C153L57697 2005
005.13'3–dc22 2004029301

ISBN 0-201-72148-1
Text printed in the United States on recycled paper at Courier Stoughton in Stoughton, Massachusetts.
11th Printing January2010

Preface

C++ Primer, Fourth Edition, provides a comprehensive introduction to the C++ language. As a primer, it provides a clear tutorial approach to the language, enhanced by numerous examples and other learning aids. Unlike most primers, it also provides a detailed description of the language, with particular emphasis on current and effective programming techniques.

Countless programmers have used previous editions of *C++ Primer* to learn C++. In that time C++ has matured greatly. Over the years, the focus of the language—and of C++ programmers—has grown beyond a concentration on runtime efficiency to focus on ways of making *programmers* more efficient. With the widespread availability of the standard library, it is possible to use and learn C++ more effectively than in the past. This revision of the *C++ Primer* reflects these new possiblities.

Changes to the Fourth Edition

In this edition, we have completely reorganized and rewritten the *C++ Primer* to highlight modern styles of C++ programming. This edition gives center stage to using the standard library while deemphasizing techniques for low-level programming. We introduce the standard library much earlier in the text and have reformulated the examples to take advantage of library facilities. We have also streamlined and reordered the presentation of language topics.

In addition to restructuring the text, we have incorporated several new elements to enhance the reader's understanding. Each chapter concludes with a Chapter Summary and glossary of Defined Terms, which recap the chapter's most important points. Readers should use these sections as a personal checklist: If you do not understand a term, restudy the corresponding part of the chapter.

We've also incorporated a number of other learning aids in the body of the text:

- Important terms are indicated in **bold**; important terms that we assume are already familiar to the reader are indicated in ***bold italics***. Each term appears in the chapter's Defined Terms section.

- Throughout the book, we highlight parts of the text to call attention to important aspects of the language, warn about common pitfalls, suggest good programming practices, and provide general usage tips. We hope that these notes will help readers more quickly digest important concepts and avoid common pitfalls.

- To make it easier to follow the relationships among features and concepts, we provide extensive forward and backward cross-references.
- We have provided sidebar discussions that focus on important concepts and supply additional explanations for topics that programmers new to C++ often find most difficult.
- Learning any programming language requires writing programs. To that end, the primer provides extensive examples throughout the text. Source code for the extended examples is available on the Web at the following URL:

```
http://www.awprofessional.com/cpp_primer
```

What hasn't changed from earlier versions is that the book remains a comprehensive tutorial introduction to C++. Our intent is to provide a clear, complete and correct guide to the language. We teach the language by presenting a series of examples, which, in addition to explaining language features, show how to make the best use of C++. Although knowledge of C (the language on which C++ was originally based) is not assumed, we do assume the reader has programmed in a modern block-structured language.

Structure of This Book

C++ Primer provides an introduction to the International Standard on C++, covering both the language proper and the extensive library that is part of that standard. Much of the power of C++ comes from its support for programming with abstractions. Learning to program effectively in C++ requires more than learning new syntax and semantics. Our focus is on how to use the features of C++ to write programs that are safe, that can be built quickly, and yet offer performance comparable to the sorts of low-level programs often written in C.

C++ is a large language and can be daunting to new users. Modern C++ can be thought of as comprising three parts:

- The low-level language, largely inherited from C
- More advanced language features that allow us to define our own data types and to organize large-scale programs and systems
- The standard library, which uses these advanced features to provide a set of useful data structures and algorithms

Most texts present C++ in this same order: They start by covering the low-level details and then introduce the the more advanced language features. They explain the standard library only after having covered the entire language. The result, all too often, is that readers get bogged down in issues of low-level programming or the complexities of writing type definitions and never really understand the power of programming in a more abstract way. Needless to say, readers also often do not learn enough to build their own abstractions.

In this edition we take a completely different tack. We start by covering the basics of the language and the library together. Doing so allows you, the reader, to

write significant programs. Only after a thorough grounding in using the library—and writing the kinds of abstract programs that the libary allows—do we move on to those features of C++ that will enable you to write your own abstractions.

Parts I and II cover the basic language and library facilities. The focus of these parts is to learn how to write C++ programs and how to use the abstractions from the library. Most C++ programmers need to know essentially everything covered in this portion of the book.

In addition to teaching the basics of C++, the material in Parts I and II serves another important purpose. The library facilities are themselves abstract data types written in C++. The library can be defined using the same class-construction features that are available to any C++ programmer. Our experience in teaching C++ is that by first using well-designed abstract types, readers find it easier to understand how to build their own types.

Parts III through V focus on how we can write our own types. Part III introduces the heart of C++: its support for classes. The class mechanism provides the basis for writing our own abstractions. Classes are also the foundation for object-oriented and generic programming, which we cover in Part IV. The *Primer* concludes with Part V, which covers advanced features that are of most use in structuring large, complex systems.

Acknowledgments

As in previous editions of this *Primer*, we'd like to extend our thanks to Bjarne Stroustrup for his tireless work on C++ and for his friendship to these authors throughout most of that time. We'd also like to thank Alex Stepanov for his original insights that led to the containers and algorithms that form the core of the standard library. Finally, our thanks go to the C++ Standards committee members for their hard work in clarifying, refining, and improving C++ over many years.

We also extend our deep-felt thanks to our reviewers, whose helpful comments on multiple drafts led us to make improvements great and small throughout the book: Paul Abrahams, Michael Ball, Mary Dageforde, Paul DuBois, Matt Greenwood, Matthew P. Johnson, Andrew Koenig, Nevin Liber, Bill Locke, Robert Murray, Phil Romanik, Justin Shaw, Victor Shtern, Clovis Tondo, Daveed Vandevoorde, and Steve Vinoski.

This book was typeset using LaTeX and the many packages that accompany the LaTeX distribution. Our well-justified thanks go to the members of the LaTeX community, who have made available such powerful typesetting tools.

The examples in this book have been compiled on the GNU and Microsoft compilers. Our thanks to their developers, and to those who have developed all the other C++ compilers, thereby making C++ a reality.

Finally, we thank the fine folks at Addison-Wesley who have shepherded this edition through the publishing process: Debbie Lafferty, our original editor, who initiated this edition and who had been with the *Primer* from its very first edition; Peter Gordon, our new editor, whose insistence on updating and streamlining the text have, we hope, greatly improved the presentation; Kim Boedigheimer, who keeps us all on schedule; and Tyrrell Albaugh, Jim Markham, Elizabeth Ryan, and John Fuller, who saw us through the design and production process.

CHAPTER 1

GETTING STARTED

CONTENTS

This chapter introduces most of the basic elements of C++: built-in, library, and class types; variables; expressions; statements; and functions. Along the way, we'll briefly explain how to compile and execute a program.

Having read this chapter and worked through the exercises, the reader should be able to write, compile, and execute simple programs. Subsequent chapters will explain in more detail the topics introduced here.

Learning a new programming language requires writing programs. In this chapter, we'll write a program to solve a simple problem that represents a common data-processing task: A bookstore keeps a file of transactions, each of which records the sale of a given book. Each transaction contains an ISBN (International Standard Book Number, a unique identifier assigned to most books published throughout the world), the number of copies sold, and the price at which each copy was sold. Each transaction looks like

```
0-201-70353-X 4 24.99
```

where the first element is the ISBN, the second is the number of books sold, and the last is the sales price. Periodically the bookstore owner reads this file and computes the number of copies of each title sold, the total revenue from that book, and the average sales price. We want to supply a program do these computations.

Before we can write this program we need to know some basic features of C++. At a minimum we'll need to know how to write, compile, and execute a simple program. What must this program do? Although we have not yet designed our solution, we know that the program must

- Define variables
- Do input and output
- Define a data structure to hold the data we're managing
- Test whether two records have the same ISBN
- Write a loop that will process every record in the transaction file

We'll start by reviewing these parts of C++ and then write a solution to our bookstore problem.

1.1 Writing a Simple C++ Program

Every C++ program contains one or more ***functions***, one of which must be named **`main`**. A function consists of a sequence of ***statements*** that perform the work of the function. The operating system executes a program by calling the function named `main`. That function executes its constituent statements and returns a value to the operating system.

Here is a simple version of `main` does nothing but return a value:

```
int main()
{
    return 0;
}
```

The operating system uses the value returned by `main` to determine whether the program succeeded or failed. A return value of 0 indicates success.

The `main` function is special in various ways, the most important of which are that the function must exist in every C++ program and it is the (only) function that the operating system explicitly calls.

We define `main` the same way we define other functions. A function definition specifies four elements: the ***return type***, the ***function name***, a (possibly empty) ***parameter list*** enclosed in parentheses, and the ***function body***. The `main` function may have only a restricted set of parameters. As defined here, the parameter list is empty; Section 7.2.6 (p. 243) will cover the other parameters that can be defined for `main`.

The `main` function is required to have a return type of `int`, which is the type that represents integers. The `int` type is a **built-in type**, which means that the type is defined by the language.

The final part of a function definition, the function body, is a ***block*** of statements starting with an open **curly brace** and ending with a close curly:

```
{
    return 0;
}
```

The only statement in our program is a `return`, which is a statement that terminates a function.

Note the semicolon at the end of the `return` statement. Semicolons mark the end of most statements in C++. They are easy to overlook, but when forgotten can lead to mysterious compiler error messages.

When the `return` includes a value such as `0`, that value is the return value of the function. The value returned must have the same type as the return type of the function or be a type that can be converted to that type. In the case of `main` the return type must be `int`, and the value `0` is an `int`.

On most systems, the return value from `main` is a status indicator. A return value of `0` indicates the successful completion of `main`. Any other return value has a meaning that is defined by the operating system. Usually a nonzero return indicates that an error occurred. Each operating system has its own way of telling the user what `main` returned.

1.1.1 Compiling and Executing Our Program

Having written the program, we need to compile it. How you compile a program depends on your operating system and compiler. For details on how your particular compiler works, you'll need to check the reference manual or ask a knowledgeable colleague.

Many PC-based compilers are run from an integrated development environment (IDE) that bundles the compiler with associated build and analysis tools. These environments can be a great asset in developing complex programs but require a fair bit of time to learn how to use effectively. Most of these environments include a point-and-click interface that allows the programmer to write a program and use various menus to compile and execute the program. Learning how to use such environments is well beyond the scope of this book.

Most compilers, including those that come with an IDE, provide a command-line interface. Unless you are already familiar with using your compiler's IDE,

it can be easier to start by using the simpler, command-line interface. Using the command-line interface lets you avoid the overhead of learning the IDE before learning the language.

Program Source File Naming Convention

Whether we are using a command-line interface or an IDE, most compilers expect that the program we want to compile will be stored in a file. Program files are referred to as **source files**. On most systems, a source file has a name that consists of two parts: a file name—for example, `prog1`—and a file suffix. By convention, the suffix indicates that the file is a program. The suffix often also indicates what language the program is written in and selects which compiler to run. The system that we used to compile the examples in this book treats a file with a suffix of `.cc` as a C++ program and so we stored this program as

```
prog1.cc
```

The suffix for C++ program files depends on which compiler you're running. Other conventions include

```
prog1.cxx
prog1.cpp
prog1.cp
prog1.C
```

Invoking the GNU or Microsoft Compilers

The command used to invoke the C++ compiler varies across compilers and operating systems. The most common compilers are the GNU compiler and the Microsoft Visual Studio compilers. By default the command to invoke the GNU compiler is `g++`:

```
$ g++ prog1.cc -o prog1
```

where `$` is the system prompt. This command generates an executable file named `prog1` or `prog1.exe`, depending on the operating system. On UNIX, executable files have no suffix; on Windows, the suffix is `.exe`. The `-o prog1` is an argument to the compiler and names the file in which to put the executable file. If the `-o prog1` is omitted, then the compiler generates an executable named `a.out` on UNIX systems and `a.exe` on Windows.

The Microsoft compilers are invoked using the command `cl`:

```
C:\directory> cl -GX prog1.cpp
```

where `C:directory>` is the system prompt and `directory` is the name of the current directory. The command to invoke the compiler is `cl`, and `-GX` is an option that is required for programs compiled using the command-line interface. The Microsoft compiler automatically generates an executable with a name that corresponds to the source file name. The executable has the suffix `.exe` and the same name as the source file name. In this case, the executable is named `prog1.exe`.

For further information consult your compiler's user's guide.

Running the Compiler from the Command Line

If we are using a command-line interface, we will typically compile a program in a console window (such as a shell window on a UNIX system or a Command Prompt window on Windows). Assuming that our `main` program is in a file named `prog1.cc`, we might compile it by using a command such as:

```
$ CC prog1.cc
```

where `CC` names the compiler and `$` represents the system prompt. The output of the compiler is an executable file that we invoke by naming it. On our system, the compiler generates the executable in a file named `a.exe`. UNIX compilers tend to put their executables in a file named `a.out`. To run an executable we supply that name at the command-line prompt:

```
$ a.exe
```

executes the program we compiled. On UNIX systems you sometimes must also specify which directory the file is in, even if it is in the current directory. In such cases, we would write

```
$ ./a.exe
```

The "`.`" followed by a slash indicates that the file is in the current directory.

The value returned from `main` is accessed in a system-dependent manner. On both UNIX and Windows systems, after executing the program, you must issue an appropriate `echo` command. On UNIX systems, we obtain the status by writing

```
$ echo $?
```

To see the status on a Windows system, we write

```
C:\directory> echo %ERRORLEVEL%
```

EXERCISES SECTION 1.1.1

Exercise 1.1: Review the documentation for your compiler and determine what file naming convention it uses. Compile and run the `main` program from page 2.

Exercise 1.2: Change the program to return `-1`. A return value of `-1` is often treated as an indicator that the program failed. However, systems vary as to how (or even whether) they report a failure from `main`. Recompile and rerun your program to see how your system treats a failure indicator from `main`.

1.2 A First Look at Input/Output

C++ does not directly define any statements to do input or output (IO). Instead, IO is provided by the **standard library**. The IO library provides an extensive set of

facilities. However, for many purposes, including the examples in this book, one needs to know only a few basic concepts and operations.

Most of the examples in this book use the **iostream library**, which handles formatted input and output. Fundamental to the iostream library are two types named **istream** and **ostream**, which represent input and output streams, respectively. A stream is a sequence of characters intended to be read from or written to an IO device of some kind. The term "stream" is intended to suggest that the characters are generated, or consumed, sequentially over time.

1.2.1 Standard Input and Output Objects

The library defines four IO objects. To handle input, we use an object of type istream named **cin** (pronounced "see-in"). This object is also referred to as the **standard input**. For output, we use an ostream object named **cout** (pronounced "see-out"). It is often referred to as the **standard output**. The library also defines two other ostream objects, named **cerr** and **clog** (pronounced "see-err" and "see-log," respectively). The cerr object, referred to as the **standard error**, is typically used to generate warning and error messages to users of our programs. The clog object is used for general information about the execution of the program.

Ordinarily, the system associates each of these objects with the window in which the program is executed. So, when we read from cin, data is read from the window in which the program is executing, and when we write to cout, cerr, or clog, the output is written to the same window. Most operating systems give us a way of redirecting the input or output streams when we run a program. Using redirection we can associate these streams with files of our choosing.

1.2.2 A Program that Uses the IO Library

So far, we have seen how to compile and execute a simple program, although that program did no work. In our overall problem, we'll have several records that refer to the same ISBN. We'll need to consolidate those records into a single total, implying that we'll need to know how to add the quantities of books sold.

To see how to solve part of that problem, let's start by looking at how we might add two numbers. Using the IO library, we can extend our main program to ask the user to give us two numbers and then print their sum:

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2;
    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1 + v2 << std::endl;
    return 0;
}
```

This program starts by printing

```
Enter two numbers:
```

on the user's screen and then waits for input from the user. If the user enters

```
3 7
```

followed by a newline, then the program produces the following output:

```
The sum of 3 and 7 is 10
```

The first line of our program is a **preprocessor directive**:

```
#include <iostream>
```

which tells the compiler that we want to use the `iostream` library. The name inside angle brackets is a **header**. Every program that uses a library facility must include its associated header. The `#include` directive must be written on a single line—the name of the header and the `#include` must appear on the same line. In general, `#include` directives should appear outside any function. Typically, all the `#include` directives for a program appear at the beginning of the file.

Writing to a Stream

The first statement in the body of `main` executes an **expression**. In C++ an expression is composed of one or more operands and (usually) an operator. The expressions in this statement use the **output operator** (the `<<` operator) to print the prompt on the standard output:

```
std::cout << "Enter two numbers:" << std::endl;
```

This statement uses the output operator twice. Each instance of the output operator takes two operands: The left-hand operand must be an `ostream` object; the right-hand operand is a value to print. The operator writes its right-hand operand to the `ostream` that is its left-hand operand.

In C++ every expression produces a result, which typically is the value generated by applying an operator to its operands. In the case of the output operator, the result is the value of its left-hand operand. That is, the value returned by an output operation is the output stream itself.

The fact that the operator returns its left-hand operand allows us to chain together output requests. The statement that prints our prompt is equivalent to

```
(std::cout << "Enter two numbers:") << std::endl;
```

Because `(std::cout << "Enter two numbers:")` returns its left operand, `std::cout`, this statement is equivalent to

```
std::cout << "Enter two numbers:";
std::cout << std::endl;
```

`endl` is a special value, called a **manipulator**, that when written to an output stream has the effect of writing a newline to the output and flushing the ***buffer*** associated with that device. By flushing the buffer, we ensure that the user will see the output written to the stream immediately.

Programmers often insert print statements during debugging. Such statements should always flush the stream. Forgetting to do so may cause output to be left in the buffer if the program crashes, leading to incorrect inferences about where the program crashed.

Using Names from the Standard Library

Careful readers will note that this program uses `std::cout` and `std::endl` rather than just `cout` and `endl`. The prefix `std::` indicates that the names `cout` and `endl` are defined inside the **namespace** named **`std`**. Namespaces allow programmers to avoid inadvertent collisions with the same names defined by a library. Because the names that the standard library defines are defined in a namespace, we can use the same names for our own purposes.

One side effect of the library's use of a namespace is that when we use a name from the library, we must say explicitly that we want to use the name from the `std` namespace. Writing `std::cout` uses the **scope operator** (the `::` operator) to say that we want to use the name `cout` that is defined in the namespace `std`. We'll see in Section 3.1 (p. 78) a way that programs often use to avoid this verbose syntax.

Reading From a Stream

Having written our prompt, we next want to read what the user writes. We start by defining two ***variables*** named `v1` and `v2` to hold the input:

```
int v1, v2;
```

We define these variables as type `int`, which is the built-in type representing integral values. These variables are ***uninitialized***, meaning that we gave them no initial value. Our first use of these variables will be to read a value into them, so the fact that they have no initial value is okay.

The next statement

```
std::cin >> v1 >> v2;
```

reads the input. The **input operator** (the `>>` operator) behaves analogously to the output operator. It takes an `istream` as its left-hand operand and an object as its right-hand operand. It reads from its `istream` operand and stores the value it read in its right-hand operand. Like the output operator, the input operator returns its left-hand operand as its result. Because the operator returns its left-hand operand, we can combine a sequence of input requests into a single statement. In other words, this input operation is equivalent to

```
std::cin >> v1;
std::cin >> v2;
```

The effect of our input operation is to read two values from the standard input, storing the first in `v1` and the second in `v2`.

Completing the Program

What remains is to print our result:

```
std::cout << "The sum of " << v1 << " and " << v2
          << " is " << v1 + v2 << std::endl;
```

This statement, although it is longer than the statement that printed the prompt, is conceptually no different. It prints each of its operands to the standard output. What is interesting is that the operands are not all the same kinds of values. Some operands are **string literals**, such as

```
"The sum of "
```

and others are various `int` values, such as `v1`, `v2`, and the result of evaluating the arithmetic expression:

```
v1 + v2
```

The `iostream` library defines versions of the input and output operators that accept all of the built-in types.

When writing a C++ program, in most places that a space appears we could instead use a newline. One exception to this rule is that spaces inside a string literal cannot be replaced by a newline. Another exception is that spaces are not allowed inside preprocessor directives.

KEY CONCEPT: INITIALIZED AND UNINITIALIZED VARIABLES

Initialization is an important concept in C++ and one to which we will return throughout this book.

Initialized variables are those that are given a value when they are defined. Uninitialized variables are not given an initial value:

```
int val1 = 0;      // initialized
int val2;          // uninitialized
```

It is almost always right to give a variable an initial value, but we are not required to do so. When we are certain that the first use of a variable gives it a new value, then there is no need to invent an initial value. For example, our first nontrivial program on page 6 defined uninitialized variables into which we immediately read values.

When we define a variable, we should give it an initial value unless we are *certain* that the initial value will be overwritten before the variable is used for any other purpose. If we cannot guarantee that the variable will be reset before being read, we should initialize it.

Exercises Section 1.2.2

Exercise 1.3: Write a program to print "Hello, World" on the standard output.

Exercise 1.4: Our program used the built-in addition operator, +, to generate the sum of two numbers. Write a program that uses the multiplication operator, *, to generate the product of two numbers.

Exercise 1.5: We wrote the output in one large statement. Rewrite the program to use a separate statement to print each operand.

Exercise 1.6: Explain what the following program fragment does:

```
std::cout << "The sum of " << v1;
          << " and " << v2;
          << " is " << v1 + v2
          << std::endl;
```

Is this code legal? If so, why? If not, why not?

1.3 A Word About Comments

Before our programs get much more complicated, we should see how C++ handles ***comments***. Comments help the human readers of our programs. They are typically used to summarize an algorithm, identify the purpose of a variable, or clarify an otherwise obscure segment of code. Comments do not increase the size of the executable program. The compiler ignores all comments.

In this book, we italicize comments to make them stand out from the normal program text. In actual programs, whether comment text is distinguished from the text used for program code depends on the sophistication of the programming environment.

There are two kinds of comments in C++: single-line and paired. A single-line comment starts with a double slash (//). Everything to the right of the slashes on the current line is a comment and ignored by the compiler.

The other delimiter, the comment pair (/**/), is inherited from the C language. Such comments begin with a /* and end with the next */. The compiler treats everything that falls between the /* and */ as part of the comment:

```
#include <iostream>
/*  Simple main function: Read two numbers and write their sum  */
int main()
{
    //  prompt user to enter two numbers
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;           //  uninitialized
    std::cin >> v1 >> v2; //  read input
    return 0;
}
```

A comment pair can be placed anywhere a tab, space, or newline is permitted. Comment pairs can span multiple lines of a program but are not required to do so. When a comment pair does span multiple lines, it is often a good idea to indicate visually that the inner lines are part of a multi-line comment. Our style is to begin each line in the comment with an asterisk, thus indicating that the entire range is part of a multi-line comment.

Programs typically contain a mixture of both comment forms. Comment pairs generally are used for multi-line explanations, whereas double slash comments tend to be used for half-line and single-line remarks.

Too many comments intermixed with the program code can obscure the code. It is usually best to place a comment block above the code it explains.

Comments should be kept up to date as the code itself changes. Programmers expect comments to remain accurate and so believe them, even when other forms of system documentation are known to be out of date. An incorrect comment is worse than no comment at all because it may mislead a subsequent reader.

Comment Pairs Do Not Nest

A comment that begins with `/*` always ends with the next `*/`. As a result, one comment pair cannot occur within another. The compiler error message(s) that result from this kind of program mistake can be mysterious and confusing. As an example, compile the following program on your system:

```
#include <iostream>
/*
 *  comment pairs /*   */ cannot nest.
 *  ''cannot nest'' is considered source code,
 *  as is the rest of the program
 */
int main()
{
    return 0;
}
```

When commenting out a large section of a program, it can seem easiest to put a comment pair around a region that you want to omit temporarily. The trouble is that if that code already has a comment pair, then the newly inserted comment will terminate prematurely. A better way to temporarily ignore a section of code is to use your editor to insert single-line comment at the beginning of each line of code you want to ignore. That way, you need not worry about whether the code you are commenting out already contains a comment pair.

1.4 Control Structures

Statements execute sequentially: The first statement in a function is executed first, followed by the second, and so on. Of course, few programs—including the one we'll need to write to solve our bookstore problem—can be written using only sequential execution. Instead, programming languages provide various control

EXERCISES SECTION 1.3

Exercise 1.7: Compile a program that has incorrectly nested comments.

Exercise 1.8: Indicate which, if any, of the following output statements, are legal.

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
```

After you've predicted what will happen, test your answer by compiling a program with these three statements. Correct any errors you encounter.

structures that allow for more complicated execution paths. This section will take a brief look at some of the control structures provided by C++. Chapter 6 covers statements in detail.

1.4.1 The `while` Statement

A **`while` statement** provides for iterative execution. We could use a `while` to write a program to sum the numbers from `1` through `10` inclusive as follows:

```
#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // keep executing the while until val is greater than 10
    while (val <= 10) {
        sum += val;  // assigns sum + val to sum
        ++val;       // add 1 to val
    }
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

This program when compiled and executed will print:

```
Sum of 1 to 10 inclusive is 55
```

As before, we begin by including the `iostream` header and define a `main` function. Inside `main` we define two `int` variables: `sum`, which will hold our summation, and `val`, which will represent each of the values from 1 through 10. We give `sum` an initial value of zero and start `val` off with the value one.

The important part is the `while` statement. A `while` has the form

```
while (condition) while_body_statement;
```

A `while` executes by (repeatedly) testing the *condition* and executing the associated *while_body_statement* until the *condition* is false.

A **condition** is an expression that is evaluated so that its result can be tested. If the resulting value is nonzero, then the condition is true; if the value is zero then the condition is false.

If the *condition* is true (the expression evaluates to a value other than zero) then *while_body_statement* is executed. After executing *while_body_statement*, the *condition* is tested again. If *condition* remains true, then the *while_body_statement* is again executed. The `while` continues, alternatively testing the *condition* and executing *while_body_statement* until the *condition* is false.

In this program, the `while` statement is:

```
// keep executing the while until val is greater than 10
while (val <= 10) {
    sum += val;  // assigns sum + val to sum
    ++val;       // add 1 to val
}
```

The condition in the `while` uses the **less-than-or-equal operator** (the <= operator) to compare the current value of `val` and `10`. As long as `val` is less than or equal to 10, we execute the body of the `while`. In this case, the body of the `while` is a **block** containing two statements:

```
{
    sum += val;  // assigns sum + val to sum
    ++val;       // add 1 to val
}
```

A block is a sequence of statements enclosed by curly braces. In C++, a block may be used wherever a statement is expected. The first statement in the block uses the **compound assignment operator**, (the += operator). This operator adds its right-hand operand to its left-hand operand. It has the same effect as writing an addition and an **assignment**:

```
sum = sum + val;  // assign sum + val to sum
```

Thus, the first statement adds the value of `val` to the current value of `sum` and stores the result back into `sum`.

The next statement

```
++val;       // add 1 to val
```

uses the **prefix increment operator** (the ++ operator). The increment operator adds one to its operand. Writing `++val` is the same as writing `val = val + 1`.

After executing the `while` body we again execute the condition in the `while`. If the (now incremented) value of `val` is still less than or equal to 10, then the body of the `while` is executed again. The loop continues, testing the condition and executing the body, until `val` is no longer less than or equal to 10.

Once `val` is greater than 10, we fall out of the `while` loop and execute the statement following the `while`. In this case, that statement prints our output, followed by the `return`, which completes our `main` program.

KEY CONCEPT: INDENTATION AND FORMATTING OF C++ PROGRAMS

C++ programs are largely free-format, meaning that the positioning of curly braces, indentation, comments, and newlines usually has no effect on the meaning of our programs. For example, the curly brace that denotes the beginning of the body of `main` could be on the same line as `main`, positioned as we have done, at the beginning of the next line, or placed anywhere we'd like. The only requirement is that it be the first nonblank, noncomment character that the compiler sees after the close parenthesis that concludes `main`'s parameter list.

Although we are largely free to format programs as we wish, the choices we make affect the readability of our programs. We could, for example, have written `main` on a single, long line. Such a definition, although legal, would be hard to read.

Endless debates occur as to the right way to format C or C++ programs. Our belief is that there is no single correct style but that there is value in consistency. We tend to put the curly braces that delimit functions on their own lines. We tend to indent compound input or output expressions so that the operators line up, as we did with the statement that wrote the output in the `main` function on page 6. Other indentation conventions will become clear as our programs become more complex.

The important thing to keep in mind is that other ways to format programs are possible. When choosing a formatting style, think about how it affects readability and comprehension. Once you've chosen a style, use it consistently.

1.4.2 The `for` Statement

In our `while` loop, we used the variable `val` to control how many times we iterated through the loop. On each pass through the `while`, the value of `val` was tested and then in the body the value of `val` was incremented.

The use of a variable like `val` to control a loop happens so often that the language defines a second control structure, called a **`for` statement**, that abbreviates the code that manages the loop variable. We could rewrite the program to sum the numbers from 1 through 10 using a `for` loop as follows:

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 up to 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val;  // equivalent to sum = sum + val

    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

Prior to the `for` loop, we define `sum`, which we set to zero. The variable `val` is used only inside the iteration and is defined as part of the `for` statement itself. The `for` statement

```
for (int val = 1; val <= 10; ++val)
    sum += val;   // equivalent to sum = sum + val
```

has two parts: the `for` header and the `for` body. The header controls how often the body is executed. The header itself consists of three parts: an *init-statement*, a *condition*, and an *expression*. In this case, the *init-statement*

```
int val = 1;
```

defines an `int` object named `val` and gives it an initial value of one. The *init-statement* is performed only once, on entry to the `for`. The *condition*

```
val <= 10
```

which compares the current value in `val` to 10, is tested each time through the loop. As long as `val` is less than or equal to 10, we execute the `for` body. Only after executing the body is the *expression* executed. In this `for`, the expression uses the prefix increment operator, which as we know adds one to the value of `val`. After executing the *expression*, the `for` retests the *condition*. If the new value of `val` is still less than or equal to `10`, then the `for` loop body is executed and `val` is incremented again. Execution continues until the *condition* fails.

In this loop, the `for` body performs the summation

```
sum += val;   // equivalent to sum = sum + val
```

The body uses the compound assignment operator to add the current value of `val` to `sum`, storing the result back into `sum`.

To recap, the overall execution flow of this `for` is:

1. Create `val` and initialize it to `1`.
2. Test whether `val` is less than or equal to `10`.
3. If `val` is less than or equal to 10, execute the `for` body, which adds `val` to `sum`. If `val` is not less than or equal to 10, then break out of the loop and continue execution with the first statement following the `for` body.
4. Increment `val`.
5. Repeat the test in step 2, continuing with the remaining steps as long as the condition is true.

When we exit the `for` loop, the variable `val` is no longer accessible. It is not possible to use `val` after this loop terminates. However, not all compilers enforce this requirement.

In pre-Standard C++ names defined in a `for` header *were* accessible outside the `for` itself. This change in the language definition can surprise people accustomed to using an older compiler when they instead use a compiler that adheres to the standard.

COMPILATION REVISITED

Part of the compiler's job is to look for errors in the program text. A compiler cannot detect whether the meaning of a program is correct, but it can detect errors in the *form* of the program. The following are the most common kinds of errors a compiler will detect.

1. **Syntax errors. The programmer has made a grammatical error in the C++ language. The following program illustrates common syntax errors; each comment describes the error on the following line:**

```
                    //  error: missing ')' in parameter list for main
int main ( {
                    //  error: used colon, not a semicolon after endl
    std::cout << "Read each file." << std::endl:
                    //  error: missing quotes around string literal
    std::cout << Update master. << std::endl;
                    //  ok: no errors on this line
    std::cout << "Write new master." << std::endl;
                    //  error: missing ';' on return statement
    return 0
}
```

2. **Type errors. Each item of data in C++ has an associated type. The value 10, for example, is an integer. The word "hello" surrounded by double quotation marks is a string literal. One example of a type error is passing a string literal to a function that expects an integer argument.**

3. **Declaration errors. Every name used in a C++ program must be declared before it is used. Failure to declare a name usually results in an error message. The two most common declaration errors are to forget to use `std::` when accessing a name from the library or to inadvertently misspell the name of an identifier:**

```
#include <iostream>
int main()
{
    int v1, v2;
    std::cin >> v >> v2;   //  error: uses "v" not "v1"
    //  cout not defined, should be std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}
```

An error message contains a line number and a brief description of what the compiler believes we have done wrong. It is a good practice to correct errors in the sequence they are reported. Often a single error can have a cascading effect and cause a compiler to report more errors than actually are present. It is also a good idea to recompile the code after each fix—or after making at most a small number of obvious fixes. This cycle is known as *edit-compile-debug*.

EXERCISES SECTION 1.4.2

Exercise 1.9: What does the following `for` loop do? What is the final value of `sum`?

```
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
```

Exercise 1.10: Write a program that uses a `for` loop to sum the numbers from 50 to 100. Now rewrite the program using a `while`.

Exercise 1.11: Write a program using a `while` loop to print the numbers from 10 down to 0. Now rewrite the program using a `for`.

Exercise 1.12: Compare and contrast the loops you wrote in the previous two exercises. Are there advantages or disadvantages to using either form?

Exercise 1.13: Compilers vary as to how easy it is to understand their diagnostics. Write programs that contain the common errors discussed in the box on 16. Study the messages the compiler generates so that these messages will be familiar when you encounter them while compiling more complex programs.

1.4.3 The `if` Statement

A logical extension of summing the values between 1 and 10 is to sum the values between two numbers our user supplies. We might use the numbers directly in our `for` loop, using the first input as the lower bound for the range and the second as the upper bound. However, if the user gives us the higher number first, that strategy would fail: Our program would exit the `for` loop immediately. Instead, we should adjust the range so that the larger number is the upper bound and the smaller is the lower. To do so, we need a way to see which number is larger.

Like most languages, C++ provides an **`if` statement** that supports conditional execution. We can use an `if` to write our revised sum program:

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; //  read input
    //  use smaller number as lower bound for summation
    //  and larger number as upper bound
    int lower, upper;
    if (v1 <= v2) {
        lower = v1;
        upper = v2;
    } else {
        lower = v2;
        upper = v1;
    }
```

```
    int sum = 0;
    // sum values from lower up to and including upper
    for (int val = lower; val <= upper; ++val)
        sum += val;  // sum = sum + val

    std::cout << "Sum of " << lower
              << " to " << upper
              << " inclusive is "
              << sum << std::endl;
    return 0;
}
```

If we compile and execute this program and give it as input the numbers 7 and 3, then the output of our program will be

```
Sum of 3 to 7 inclusive is 25
```

Most of the code in this program should already be familiar from our earlier examples. The program starts by writing a prompt to the user and defines four `int` variables. It then reads from the standard input into `v1` and `v2`. The only new code is the `if` statement

```
// use smaller number as lower bound for summation
// and larger number as upper bound
int lower, upper;
if (v1 <= v2) {
    lower = v1;
    upper = v2;
} else {
    lower = v2;
    upper = v1;
}
```

The effect of this code is to set `upper` and `lower` appropriately. The `if` condition tests whether `v1` is less than or equal to `v2`. If so, we perform the block that immediately follows the condition. This block contains two statements, each of which does an assignment. The first statement assigns `v1` to `lower` and the second assigns `v2` to `upper`.

If the condition is false—that is, if `v1` is larger than `v2`—then we execute the statement following the `else`. Again, this statement is a block consisting of two assignments. We assign `v2` to `lower` and `v1` to `upper`.

1.4.4 Reading an Unknown Number of Inputs

Another change we might make to our summation program on page 12 would be to allow the user to specify a set of numbers to sum. In this case we can't know how many numbers we'll be asked to add. Instead, we want to keep reading numbers until the program reaches the end of the input. When the input is finished, the program writes the total to the standard output:

EXERCISES SECTION 1.4.3

Exercise 1.14: What happens in the program presented in this section if the input values are equal?

Exercise 1.15: Compile and run the program from this section with two equal values as input. Compare the output to what you predicted in the previous exercise. Explain any discrepancy between what happened and what you predicted.

Exercise 1.16: Write a program to print the larger of two inputs supplied by the user.

Exercise 1.17: Write a program to ask the user to enter a series of numbers. Print a message saying how many of the numbers are negative numbers.

```
#include <iostream>
int main()
{
    int sum = 0, value;
    //  read till end-of-file, calculating a running total of all values read
    while (std::cin >> value)
        sum += value; //  equivalent to sum = sum + value
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

If we give this program the input

```
3 4 5 6
```

then our output will be

```
Sum is: 18
```

As usual, we begin by including the necessary headers. The first line inside `main` defines two `int` variables, named `sum` and `value`. We'll use `value` to hold each number we read, which we do inside the condition in the `while`:

```
while (std::cin >> value)
```

What happens here is that to evaluate the condition, the input operation

```
std::cin >> value
```

is executed, which has the effect of reading the next number from the standard input, storing what was read in `value`. The input operator (Section 1.2.2, p. 8) returns its left operand. The condition tests that result, meaning it tests `std::cin`.

When we use an `istream` as a condition, the effect is to test the state of the stream. If the stream is valid—that is, if it is still possible to read another input—then the test succeeds. An `istream` becomes invalid when we hit ***end-of-file*** or encounter an invalid input, such as reading a value that is not an integer. An `istream` that is in an invalid state will cause the condition to fail.

Until we do encounter end-of-file (or some other input error), the test will succeed and we'll execute the body of the `while`. That body is a single statement that uses the compound assignment operator. This operator adds its right-hand operand into the left hand operand.

Entering an End-of-File from the Keyboard

Operating systems use different values for end-of-file. On Windows systems we enter an end-of-file by typing a control-z—simultaneously type the "ctrl" key and a "z." On UNIX systems, including Mac OS-X machines, it is usually control-d.

Once the test fails, the `while` terminates and we fall through and execute the statement following the `while`. That statement prints `sum` followed by `endl`, which prints a newline and flushes the buffer associated with `cout`. Finally, we execute the `return`, which as usual returns zero to indicate success.

Exercises Section 1.4.4

Exercise 1.18: Write a program that prompts the user for two numbers and writes each number in the range specified by the two numbers to the standard output.

Exercise 1.19: What happens if you give the numbers 1000 and 2000 to the program written for the previous exercise? Revise the program so that it never prints more than 10 numbers per line.

Exercise 1.20: Write a program to sum the numbers in a user-specified range, omitting the `if` test that sets the upper and lower bounds. Predict what happens if the input is the numbers 7 and 3, in that order. Now run the program giving it the numbers 7 and 3, and see if the results match your expectation. If not, restudy the discussion on the `for` and `while` loop until you understand what happened.

1.5 Introducing Classes

The only remaining feature we need to understand before solving our bookstore problem is how to write a ***data structure*** to represent our transaction data. In C++ we define our own data structure by defining a **class**. The class mechanism is one of the most important features in C++. In fact, a primary focus of the design of C++ is to make it possible to define **class types** that behave as naturally as the built-in types themselves. The library types that we've seen already, such as `istream` and `ostream`, are all defined as classes—that is, they are not strictly speaking part of the language.

Complete understanding of the class mechanism requires mastering a lot of information. Fortunately, it is possible to use a class that someone else has written without knowing how to define a class ourselves. In this section, we'll describe a simple class that we can use in solving our bookstore problem. We'll implement

this class in the subsequent chapters as we learn more about types, expressions, statements, and functions—all of which are used in defining classes.

To use a class we need to know three things:

1. What is its name?
2. Where is it defined?
3. What operations does it support?

For our bookstore problem, we'll assume that the class is named `Sales_item` and that it is defined in a header named `Sales_item.h`.

1.5.1 The `Sales_item` Class

The purpose of the `Sales_item` class is to store an ISBN and keep track of the number of copies sold, the revenue, and average sales price for that book. How these data are stored or computed is not our concern. To use a class, we need not know anything about how it is implemented. Instead, what we need to know is what operations the class provides.

As we've seen, when we use library facilities such as IO, we must include the associated headers. Similarly, for our own classes, we must make the definitions associated with the class available to the compiler. We do so in much the same way. Typically, we put the class definition into a file. Any program that wants to use our class must include that file.

Conventionally, class types are stored in a file with a name that, like the name of a program source file, has two parts: a file name and a file suffix. Usually the file name is the same as the class defined in the header. The suffix usually is `.h`, but some programmers use `.H`, `.hpp`, or `.hxx`. Compilers usually aren't picky about header file names, but IDEs sometimes are. We'll assume that our class is defined in a file named `Sales_item.h`.

Operations on `Sales_item` Objects

Every class defines a type. The type name is the same as the name of the class. Hence, our `Sales_item` class defines a type named `Sales_item`. As with the built-in types, we can define a variable of a class type. When we write

```
Sales_item item;
```

we are saying that `item` is an object of type `Sales_item`. We often contract the phrase "an object of type `Sales_item`" to "a `Sales_item` object" or even more simply to "a `Sales_item`."

In addition to being able to define variables of type `Sales_item`, we can perform the following operations on `Sales_item` objects:

- Use the addition operator, `+`, to add two `Sales_item`s
- Use the input operator, `>>`, to read a `Sales_item` object

- Use the output operator, `<<`, to write a `Sales_item` object
- Use the assignment operator, `=`, to assign one `Sales_item` object to another
- Call the `same_isbn` function to determine if two `Sales_items` refer to the same book

Reading and Writing `Sales_items`

Now that we know the operations that the class provides, we can write some simple programs to use this class. For example, the following program reads data from the standard input, uses that data to build a `Sales_item` object, and writes that `Sales_item` object back onto the standard output:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    // read ISBN, number of copies sold, and sales price
    std::cin >> book;
    // write ISBN, number of copies sold, total revenue, and average price
    std::cout << book << std::endl;
    return 0;
}
```

If the input to this program is

```
0-201-70353-X 4 24.99
```

then the output will be

```
0-201-70353-X 4 99.96 24.99
```

Our input said that we sold four copies of the book at $24.99 each, and the output indicates that the total sold was four, the total revenue was $99.96, and the average price per book was $24.99.

This program starts with two `#include` directives, one of which uses a new form. The `iostream` header is defined by the standard library; the `Sales_item` header is not. `Sales_item` is a type that we ourselves have defined. When we use our own headers, we use quotation marks (`" "`) to surround the header name.

Headers for the standard library are enclosed in angle brackets (`< >`). Nonstandard headers are enclosed in double quotes (`" "`).

Inside `main` we start by defining an object, named `book`, which we'll use to hold the data that we read from the standard input. The next statement reads into that object, and the third statement prints it to the standard output followed as usual by printing `endl` to flush the buffer.

KEY CONCEPT: CLASSES DEFINE BEHAVIOR

As we go through these programs that use `Sales_items`, the important thing to keep in mind is that the author of the `Sales_item` class defined *all* the actions that can be performed by objects of this class. That is, the author of the `Sales_item` data structure defines what happens when a `Sales_item` object is created and what happens when the addition or the input and output operators are applied to `Sales_item` objects, and so on.

In general, only the operations defined by a class can be used on objects of the class type. For now, the only operations we know we can peeform on `Sales_item` objects are the ones listed on page 21.

We'll see how these operations are defined in Sections 7.7.3 and 14.2.

Adding `Sales_items`

A slightly more interesting example adds two `Sales_item` objects:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;    // read a pair of transactions
    std::cout << item1 + item2 << std::endl; // print their sum
    return 0;
}
```

If we give this program the following input

```
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
```

our output is

```
0-201-78345-X 5 110 22
```

This program starts by including the `Sales_item` and `iostream` headers. Next we define two `Sales_item` objects to hold the two transactions that we wish to sum. The output expression does the addition and prints the result. We know from the list of operations on page 21 that adding two `Sales_items` together creates a new object whose ISBN is that of its operands and whose number sold and revenue reflect the sum of the corresponding values in its operands. We also know that the items we add must represent the same ISBN.

It's worth noting how similar this program looks to the one on page 6: We read two inputs and write their sum. What makes it interesting is that instead of reading and printing the sum of two integers, we're reading and printing the sum of two `Sales_item` objects. Moreover, the whole idea of "sum" is different. In the case of `int`s we are generating a conventional sum—the result of adding two numeric values. In the case of `Sales_item` objects we use a conceptually new meaning for sum—the result of adding the components of two `Sales_item` objects.

Exercises Section 1.5.1

Exercise 1.21: The Web site (`http://www.awprofessional.com/cpp_primer`) contains a copy of `Sales_item.h` in the Chapter 1 code directory. Copy that file to your working directory. Write a program that loops through a set of book sales transactions, reading each transaction and writing that transaction to the standard output.

Exercise 1.22: Write a program that reads two `Sales_item` objects that have the same ISBN and produces their sum.

Exercise 1.23: Write a program that reads several transactions for the same ISBN. Write the sum of all the transactions that were read.

1.5.2 A First Look at Member Functions

Unfortunately, there is a problem with the program that adds `Sales_item`s. What should happen if the input referred to two different ISBNs? It doesn't make sense to add the data for two different ISBNs together. To solve this problem, we'll first check whether the `Sales_item` operands refer to the same ISBNs:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;
    // first check that item1 and item2 represent the same book
    if (item1.same_isbn(item2)) {
        std::cout << item1 + item2 << std::endl;
        return 0;   // indicate success
    } else {
        std::cerr << "Data must refer to same ISBN"
                  << std::endl;
        return -1;  // indicate failure
    }
}
```

The difference between this program and the previous one is the `if` test and its associated `else` branch. Before explaining the `if` condition, we know that what this program does depends on the condition in the `if`. If the test succeeds, then we write the same output as the previous program and return 0 indicating success. If the test fails, we execute the block following the `else`, which prints a message and returns an error indicator.

What Is a Member Function?

The `if` condition

```
// first check that item1 and item2 represent the same book
if (item1.same_isbn(item2)) {
```

calls a **member function** of the `Sales_item` object named `item1`. A member function is a function that is defined by a class. Member functions are sometimes referred to as the **methods** of the class.

Member functions are defined once for the class but are treated as members of each object. We refer to these operations as member functions because they (usually) operate on a specific object. In this sense, they are members of the object, even though a single definition is shared by all objects of the same type.

When we call a member function, we (usually) specify the object on which the function will operate. This syntax uses the **dot operator** (the "." operator):

```
item1.same_isbn
```

means "the `same_isbn` member of the object named `item1`." The dot operator fetches its right-hand operand from its left. The dot operator applies only to objects of class type: The left-hand operand must be an object of class type; the right-hand operand must name a member of that type.

Unlike most other operators, the right operand of the dot (" . ") operator is not an object or value; it is the name of a member.

When we use a member function as the right-hand operand of the dot operator, we usually do so to call that function. We execute a member function in much the same way as we do any function: To call a function, we follow the function name by the **call operator** (the "`()`" operator). The call operator is a pair of parentheses that encloses a (possibly empty) list of ***arguments*** that we pass to the function.

The `same_isbn` function takes a single argument, and that argument is another `Sales_item` object. The call

```
item1.same_isbn(item2)
```

passes `item2` as an argument to the function named `same_isbn` that is a member of the object named `item1`. This function compares the ISBN part of its argument, `item2`, to the ISBN in `item1`, the object on which `same_isbn` is called. Thus, the effect is to test whether the two objects refer to the same ISBN.

If the objects refer to the same ISBN, we execute the statement following the `if`, which prints the result of adding the two `Sales_item` objects together. Otherwise, if they refer to different ISBNs, we execute the `else` branch, which is a block of statements. The block prints an appropriate error message and exits the program, returning `-1`. Recall that the return from `main` is treated as a status indicator. In this case, we return a nonzero value to indicate that the program failed to produce the expected result.

1.6 The C++ Program

Now we are ready to solve our original bookstore problem: We need to read a file of sales transactions and produce a report that shows for each book the total revenue, average sales price, and the number of copies sold.

Exercises Section 1.5.2

Exercise 1.24: Write a program that reads several transactions. For each new transaction that you read, determine if it is the same ISBN as the previous transaction, keeping a count of how many transactions there are for each ISBN. Test the program by giving multiple transactions. These transactions should represent multiple ISBNs but the records for each ISBN should be grouped together.

We'll assume that all of the transactions for a given ISBN appear together. Our program will combine the data for each ISBN in a `Sales_item` object named `total`. Each transaction we read from the standard input will be stored in a second `Sales_item` object named `trans`. Each time we read a new transaction we'll compare it to the `Sales_item` object in `total`. If the objects refer to the same ISBN, we'll update `total`. Otherwise we'll print the value in `total` and reset it using the transaction we just read.

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    // declare variables to hold running sum and data for the next record
    Sales_item total, trans;
    // is there data to process?
    if (std::cin >> total) {
        // if so, read the transaction records
        while (std::cin >> trans)
            if (total.same_isbn(trans))
                // match: update the running total
                total = total + trans;
            else {
                // no match: print & assign to total
                std::cout << total << std::endl;
                total = trans;
            }
        // remember to print last record
        std::cout << total << std::endl;
    } else {
        // no input!, warn the user
        std::cout << "No data?!" << std::endl;
        return -1;  // indicate failure
    }
    return 0;
}
```

This program is the most complicated one we've seen so far, but it uses only facilities that we have already encountered. As usual, we begin by including the headers that we use: `iostream` from the library and `Sales_item.h`, which is our own header.

Inside `main` we define the objects we need: `total`, which we'll use to sum the data for a given ISBN, and `trans`, which will hold our transactions as we read them. We start by reading a transaction into `total` and testing whether the read was successful. If the read fails, then there are no records and we fall through to the outermost `else` branch, which prints a message to warn the user that there was no input.

Assuming we have successfully read a record, we execute the code in the `if` branch. The first statement is a `while` that will loop through all the remaining records. Just as we did in the program on page 18, our `while` condition reads a value from the standard input and then tests that valid data was actually read. In this case, we read a `Sales_item` object into `trans`. As long as the read succeeds, we execute the body of the `while`.

The body of the `while` is a single `if` statement. We test whether the ISBNs are equal, and if so we add the two objects and store the result in `total`. If the ISBNs are not equal, we print the value stored in `total` and reset `total` by assigning `trans` to it. After execution of the `if`, we return to the condition in the `while`, reading the next transaction and so on until we run out of records.

Once the `while` completes, we still must write the data associated with the last ISBN. When the `while` terminates, `total` contains the data for the last ISBN in the file, but we had no chance to print it. We do so in the last statement of the block that concludes the outermost `if` statement.

Exercises Section 1.6

Exercise 1.25: Using the `Sales_item.h` header from the Web site, compile and execute the bookstore program presented in this section.

Exercise 1.26: In the bookstore program we used the addition operator and not the compound assignment operator to add `trans` to `total`. Why didn't we use the compound assignment operator?

Chapter Summary

This chapter introduced enough of C++ to let the reader compile and execute simple C++ programs. We saw how to define a `main` function, which is the function that is executed first in any C++ program. We also saw how to define variables, how to do input and output, and how to write `if`, `for`, and `while` statements. The chapter closed by introducing the most fundamental facility in C++: the class. In this chapter we saw how to create and use objects of a given `class`. Later chapters show how to define our own classes.

Defined Terms

argument A value passed to a function when it is called.

block Sequence of statements enclosed in curly braces.

buffer A region of storage used to hold data. IO facilities often store input (or output) in a buffer and read or write the buffer independently of actions in the program. Output buffers usually must be explicitly flushed to force the buffer to be written. By default, reading `cin` flushes `cout`; `cout` is also flushed when the program ends normally.

built-in type A type, such as `int`, defined by the language.

cerr `ostream` object tied to the standard error, which is often the same stream as the standard output. By default, writes to `cerr` are not buffered. Usually used for error messages or other output that is not part of the normal logic of the program.

cin `istream` object used to read from the standard input.

class C++ mechanism for defining our own data structures. The class is one of the most fundamental features in C++. Library types, such as `istream` and `ostream`, are classes.

class type A type defined by a class. The name of the type is the class name.

clog `ostream` object tied to the standard error. By default, writes to `clog` are buffered. Usually used to report information about program execution to a log file.

comments Program text that is ignored by the compiler. C++ has two kinds of comments: single-line and paired. Single-line comments start with a `//`. Everything from the `//` to the end of the line is a comment. Paired comments begin with a `/*` and include all text up to the next `*/`.

condition An expression that is evaluated as true or false. An arithmetic expression that evaluates to zero is false; any other value yields true.

cout `ostream` object used to write to the standard output. Ordinarily used to write the output of a program.

curly brace Curly braces delimit blocks. An open curly ({) starts a block; a close curly (}) ends one.

data structure A logical grouping of data and operations on that data.

edit-compile-debug The process of getting a program to execute properly.

end-of-file System-specific marker in a file that indicates that there is no more input in the file.

expression The smallest unit of computation. An expression consists of one or more operands and usually an operator. Expressions are evaluated to produce a result. For example, assuming `i` and `j` are `int`s, then `i + j` is an arithmetic addition expression

and yields the sum of the two `int` values. Expressions are covered in more detail in Chapter 5.

for statement Control statement that provides iterative execution. Often used to step through a data structure or to repeat a calculation a fixed number of times.

function A named unit of computation.

function body Statement block that defines the actions performed by a function.

function name Name by which a function is known and can be called.

header A mechanism whereby the definitions of a class or other names may be made available to multiple programs. A header is included in a program through a `#include` directive.

if statement Conditional execution based on the value of a specified condition. If the condition is true, the `if` body is executed. If not, control flows to the statement following the `else` if there is one or to the statement following the `if` if there is no `else`.

iostream library type providing stream-oriented input and output.

istream Library type providing stream-oriented input.

library type A type, such as `istream`, defined by the standard library.

main function Function called by the operating system when executing a C++ program. Each program must have one and only one function named `main`.

manipulator Object, such as `std::endl`, that when read or written "manipulates" the stream itself. Section A.3.1 (p. 825) covers manipulators in more detail.

member function Operation defined by a class. Member functions ordinarily are called to operate on a specific object.

method Synonym for member function.

namespace Mechanism for putting names defined by a library into a single place. Namespaces help avoid inadvertent name clashes. The names defined by the C++ library are in the namespace `std`.

ostream Library type providing stream-oriented output.

parameter list Part of the definition of a function. Possibly empty list that specifies what arguments can be used to call the function.

preprocessor directive An instruction to the C++ preprocessor. `#include` is a preprocessor directive. Preprocessor directives must appear on a single line. We'll learn more about the preprocessor in Section 2.9.2.

return type Type of the value returned by a function.

source file Term used to describe a file that contains a C++ program.

standard error An output stream intended for use for error reporting. Ordinarily, on a windowing operating system, the standard output and the standard error are tied to the window in which the program is executed.

standard input The input stream that ordinarily is associated by the operating system with the window in which the program executes.

standard library Collection of types and functions that every C++ compiler must support. The library provides a rich set of capabilities including the types that support IO. C++ programmers tend to talk about "the library," meaning the entire standard library or about particular parts of the library by referring to a library type. For example, programmers also refer to the "`iostream` library," meaning the part of the standard library defined by the `iostream` classes.

standard output The output stream that ordinarily is associated by the operating system with the window in which the program executes.

statement The smallest independent unit in a C++ program. It is analogous to a sentence in a natural language. Statements in C++ generally end in semicolons.

std Name of the namespace used by the standard library. `std::cout` indicates that we're using the name `cout` defined in the `std` namespace.

string literal Sequence of characters enclosed in double quotes.

uninitialized variable Variable that has no initial value specified. There are no uninitialized variables of class type. Variables of class type for which no initial value is specified are initialized as specified by the class definition. You must give a value to an uninitialized variable before attempting to use the variable's value. *Uninitialized variables can be a rich source of bugs*.

variable A named object.

while statement An iterative control statement that executes the statement that is the `while` body as long as a specified condition is true. The body is executed zero or more times, depending on the truth value of the condition.

`()` operator The call operator: A pair of parentheses "`()`" following a function name. The operator causes a function to be invoked. Arguments to the function may be passed inside the parentheses.

`++` operator Increment operator. Adds one to the operand; `++i` is equivalent to `i = i + 1`.

`+=` operator A compound assignment operator. Adds right-hand operand to the left and stores the result back into the left-hand operand; `a += b` is equivalent to `a = a + b`.

`.` operator Dot operator. Takes two operands: the left-hand operand is an object and the right is the name of a member of that object. The operator fetches that member from the named object.

`::` operator Scope operator. We'll see more about scope in Chapter 2. Among other uses, the scope operator is used to access names in a namespace. For example, `std::cout` says to use the name `cout` from the namespace `std`.

`=` operator Assigns the value of the right-hand operand to the object denoted by the left-hand operand.

`<<` operator Output operator. Writes the right-hand operand to the output stream indicated by the left-hand operand: `cout << "hi"` writes `hi` to the standard output. Output operations can be chained together: `cout << "hi << "bye"` writes `hibye`.

`>>` operator Input operator. Reads from the input stream specified by the left-hand operand into the right-hand operand: `cin >> i` reads the next value on the standard input into `i`. Input operations can be chained together: `cin >> i >> j` reads first into `i` and then into `j`.

`==` operator The equality operator. Tests whether the left-hand operand is equal to the right-hand.

`!=` operator Assignment operator. Tests whether the left-hand operand is not equal to the right-hand.

`<=` operator The less-than-or-equal operator. Tests whether the left-hand operand is less than or equal to the right-hand.

`<` operator The less-than operator. Tests whether the left-hand operand is less than the right-hand.

`>=` operator Greater-than-or-equal operator. Tests whether the left-hand operand is greater than or equal to the right-hand.

`>` operator Greater-than operator. Tests whether the left-hand operand is greater than the right-hand.

Index

Bold face numbers refer to the page on which the term was first defined.
Numbers in *italic* refer to the "Defined Terms" section in which the term is defined.

A

C

D

E

F

K

M

N

O

P

Q

R

S

T

U

W

X

Z