Bestseller Since 1986

Completely Rewritten for the New C++11 Standard

Fifth Edition

C++ Primer

Stanley B. Lippman
Josée Lajoie
Barbara E. Moo

C++ Primer, Fifth Edition

Stanley B. Lippman
Josée Lajoie
Barbara E. Moo

Addison-Wesley
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sidney • Tokyo • Singapore • Mexico City

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Lippman, Stanley B.
C++ primer / Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. – 5th ed.
p. cm.
Includes index.
ISBN 0-321-71411-3 (pbk. : alk. paper) 1. C++ (Computer program language) I. Lajoie, Josée. II.
Moo, Barbara E. III. Title.
QA76.73.C153L57697 2013
005.13'3–
dc23 2012020184

ISBN-13: 978-0-321-71411-4
ISBN-10: 0-321-71411-3

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

First printing, August 2012

To Beth, who makes this, and all things, possible.

——

To Daniel and Anna, who contain virtually all possibilities.

—SBL

To Mark and Mom, for their unconditional love and support.

—JL

To Andy, who taught me to program and so much more.
—BEM

Contents

New Features in C++11

Preface

Countless programmers have learned C++ from previous editions of *C++ Primer*. During that time, C++ has matured greatly: Its focus, and that of its programming community, has widened from looking mostly at *machine* efficiency to devoting more attention to *programmer* efficiency.

In 2011, the C++ standards committee issued a major revision to the ISO C++ standard. This revised standard is latest step in C++'s evolution and continues the emphasis on programmer efficiency. The primary goals of the new standard are to

- Make the language more uniform and easier to teach and to learn
- Make the standard libraries easier, safer, and more efficient to use
- Make it easier to write efficient abstractions and libraries

In this edition, we have completely revised the *C++ Primer* to use the latest standard. You can get an idea of how extensively the new standard has affected C++ by reviewing the New Features Table of Contents, which lists the sections that cover new material and appears on page xxi.

Some additions in the new standard, such as `auto` for type inference, are pervasive. These facilities make the code in this edition easier to read and to understand. Programs (and programmers!) can ignore type details, which makes it easier to concentrate on what the program is intended to do. Other new features, such as smart pointers and move-enabled containers, let us write more sophisticated classes without having to contend with the intricacies of resource management. As a result, we can start to teach how to write your own classes much earlier in the book than we did in the Fourth Edition. We—and you—no longer have to worry about many of the details that stood in our way under the previous standard.

C++ 11

We've marked those parts of the text that cover features defined by the new standard, with a marginal icon. We hope that readers who are already familiar with the core of C++ will find these alerts useful in deciding where to focus their attention. We also expect that these icons will help explain error messages from compilers that might not yet support every new feature. Although nearly all of the examples in this book have been compiled under the current release of the GNU compiler, we realize some readers will not yet have access to completely updated compilers. Even though numerous capabilities have been added by the latest standard, the core language remains unchanged and forms the bulk of the material that we cover. Readers can use these icons to note which capabilities may not yet be available in their compiler.

Why Read This Book?

Modern C++ can be thought of as comprising three parts:

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

Most texts present C++ in the order in which it evolved. They teach the C subset of C++ first, and present the more abstract features of C++ as advanced topics at the end of the book. There are two problems with this approach: Readers can get bogged down in the details inherent in low-level programming and give up in frustration. Those who do press on learn bad habits that they must unlearn later.

We take the opposite approach: Right from the start, we use the features that let programmers ignore the details inherent in low-level programming. For example, we introduce and use the library `string` and `vector` types along with the built-in arithmetic and array types. Programs that use these library types are easier to write, easier to understand, and much less error-prone.

Too often, the library is taught as an “advanced” topic. Instead of using the library, many books use low-level programming techniques based on pointers to character arrays and dynamic memory management. Getting programs that use these low-level techniques to work correctly is much harder than writing the corresponding C++ code using the library.

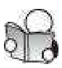
Throughout *C++ Primer*, we emphasize good style: We want to help you, the reader, develop good habits immediately and avoid needing to unlearn bad habits as you gain more sophisticated knowledge. We highlight particularly tricky matters and warn about common misconceptions and pitfalls.

We also explain the rationale behind the rules—explaining the why not just the what. We believe that by understanding why things work as they do, readers can more quickly cement their grasp of the language.

Although you do not need to know C in order to understand this book, we assume you know enough about programming to write, compile, and run a program in at least one modern block-structured language. In particular, we assume you have used variables, written and called functions, and used a compiler.

Changes to the Fifth Edition

New to this edition of *C++ Primer* are icons in the margins to help guide the reader. C++ is a large language that offers capabilities tailored to particular kinds of programming problems. Some of these capabilities are of great import for large project teams but might not be necessary for smaller efforts. As a result, not every programmer needs to know every detail of every feature. We've added these marginal icons to help the reader know which parts can be learned later and which topics are more essential.



We've marked sections that cover the fundamentals of the language with an image of a person studying a book. The topics covered in sections marked this way form the core part of the language. Everyone should read and understand these sections.

We've also indicated those sections that cover advanced or special-purpose topics. These sections can be skipped or skimmed on a first reading. We've marked such sections with a stack of books to indicate that you can safely put down the book at that point. It is probably a good idea to skim such sections so you know that the capability exists. However, there is no reason to spend time studying these topics until you actually need to use the feature in your own programs.

To help readers guide their attention further, we've noted particularly tricky concepts with a magnifying-glass icon. We hope that readers will take the time to understand thoroughly the material presented in the sections so marked. In at least some of these sections, the import of the topic may not be readily apparent; but we think you'll find that these sections cover topics that turn out to be essential to understanding the language.

Another aid to reading this book, is our extensive use of cross-references. We hope these references will make it easier for readers to dip into the middle of the book, yet easily jump back to the earlier material on which later examples rely.

What remains unchanged is that *C++ Primer* is a clear, correct, and thorough tutorial guide to C++. We teach the language by presenting a series of increasingly sophisticated examples, which explain language features and show how to make the best use of C++.

Structure of This Book

We start by covering the basics of the language and the library together in Parts I and

II. These parts cover enough material to let you, the reader, write significant programs. 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: By using the abstract facilities defined by the library, you will become more comfortable with using high-level programming techniques. The library facilities are themselves abstract data types that are usually 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.

Only after a thorough grounding in using the library—and writing the kinds of abstract programs that the library allows—do we move on to those C++ features that will enable you to write your own abstractions. Parts III and IV focus on writing abstractions in the form of classes. Part III covers the fundamentals; Part IV covers more specialized facilities.

In Part III, we cover issues of copy control, along with other techniques to make classes that are as easy to use as the built-in types. Classes are the foundation for object-oriented and generic programming, which we also cover in Part III. *C++ Primer* concludes with Part IV, which covers features that are of most use in structuring large, complicated systems. We also summarize the library algorithms in Appendix A.

Aids to the Reader

Each chapter concludes with a summary, followed by a glossary of defined terms, which together 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.
- To make it easier to follow the relationships among features and concepts, we provide extensive forward and backward cross-references.
- We provide sidebar discussions on important concepts and for topics that new C++ programmers 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.informit.com/title/032174113`

A Note about Compilers

As of this writing (July, 2012), compiler vendors are hard at work updating their compilers to match the latest ISO standard. The compiler we use most frequently is the GNU compiler, version 4.7.0. There are only a few features used in this book that this compiler does not yet implement: inheriting constructors, reference qualifiers for member functions, and the regular-expression library.

Acknowledgments

In preparing this edition we are very grateful for the help of several current and former members of the standardization committee: Dave Abrahams, Andy Koenig, Stephan T. Lavavej, Jason Merrill, John Spicer, and Herb Sutter. They provided invaluable assistance to us in understanding some of the more subtle parts of the new standard. We'd also like to thank the many folks who worked on updating the GNU compiler making the standard a reality.

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

We extend our deep-felt thanks to our reviewers, whose helpful comments led us to make improvements great and small throughout the book: Marshall Clow, Jon Kalb, Nevin Liber, Dr. C. L. 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.

Finally, we thank the fine folks at Addison-Wesley who have shepherded this edition through the publishing process: Peter Gordon, our editor, who provided the impetus for us to revise *C++ Primer* once again; Kim Boedigheimer, who keeps us all on schedule; Barbara Wood, who found lots of editing errors for us during the copy-edit phase, and Elizabeth Ryan, who was again a delight to work with as she guided us through the design and production process.

Chapter 1. Getting Started

Contents

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

After having read this chapter and worked through the exercises, you should be able to write, compile, and execute simple programs. Later chapters will assume that you can use the features introduced in this chapter, and will explain these features in more detail.

The way to learn a new programming language is to write programs. In this chapter, we'll write a program to solve a simple problem for a bookstore.

Our store keeps a file of transactions, each of which records the sale of one or more copies of a single book. Each transaction contains three data elements:

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

The first element is an ISBN (International Standard Book Number, a unique book identifier), the second is the number of copies sold, and the last is the price at which each of these copies was sold. From time to time, the bookstore owner reads this file and for each book computes the number of copies sold, the total revenue from that book, and the average sales price.

To be able to write this program, we need to cover a few basic C++ features. In addition, we'll need to know how to compile and execute a program.

Although we haven't yet designed our program, it's easy to see that it must

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

We'll start by reviewing how to solve these subproblems in C++ and then write our bookstore program.

1.1. Writing a Simple C++ Program

Every C++ program contains one or more *functions*, one of which must be named **main**. The operating system runs a C++ program by calling `main`. Here is a simple version of `main` that does nothing but return a value to the operating system:

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

A function definition has four elements: a *return type*, a *function name*, a (possibly empty) *parameter list* enclosed in parentheses, and a *function body*. Although `main` is special in some ways, we define `main` the same way we define any other function.

In this example, `main` has an empty list of parameters (shown by the `()` with nothing inside). § 6.2.5 (p. 218) will discuss the other parameter types that we can define for `main`.

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

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 this block is a `return`, which is a statement that terminates a function. As is the case here, a `return` can also send a value back to the function's caller. When a `return` statement includes a value, the value returned must have a type that is compatible with the return type of the function. In this case, the return type of `main` is `int` and the return value is `0`, which is an `int`.

Note

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.

On most systems, the value returned from `main` is a status indicator. A return value of `0` indicates success. A nonzero return has a meaning that is defined by the system.

Ordinarily a nonzero return indicates what kind of error occurred.

Key Concept: Types

Types are one of the most fundamental concepts in programming and a concept that we will come back to over and over in this Primer. A type defines both the contents of a data element and the operations that are possible on those data.

The data our programs manipulate are stored in variables and every variable has a type. When the type of a variable named `v` is `T`, we often say that "`v` has type `T`" or, interchangeably, that "`v` is a `T`."

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, 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 build and analysis tools. These environments can be a great asset in developing large programs but require a fair bit of time to learn how to use effectively. 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 already know the IDE, you may find it easier to start with the command-line interface. Doing so will let you concentrate on learning C++ first. Moreover, once you understand the language, the IDE is likely to be easier to learn.

Program Source File Naming Convention

Whether you use a command-line interface or an IDE, most compilers expect program source code to be stored in one or more files. Program files are normally referred to as a *source files*. On most systems, the name of a source file ends with a suffix, which is a period followed by one or more characters. The suffix tells the system that the file is a C++ program. Different compilers use different suffix conventions; the most common include `.cc`, `.cxx`, `.cpp`, `.cp`, and `.C`.

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 `$` is the system prompt. The compiler generates an executable file. On a Windows system, that executable file is named `prog1.exe`. UNIX compilers tend to put their executables in files named `a.out`.

To run an executable on Windows, we supply the executable file name and can omit the `.exe` file extension:

$ prog1

On some systems you must specify the file's location explicitly, even if the file is in the current directory or folder. In such cases, we would write

$.\prog1

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

To run an executable on UNIX, we use the full file name, including the file extension:

$ a.out

If we need to specify the file's location, we'd use a "`.`" followed by a forward slash to indicate that our executable is in the current directory:

$ ./a.out

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

$ echo %ERRORLEVEL%

Running the GNU or Microsoft Compilers

The command used to run 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 run the GNU compiler is `g++`:

Click here to view code image

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

Here `$` is the system prompt. The `-o prog1` is an argument to the compiler and names the file in which to put the executable file. 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`. If the `-o prog1` is omitted, the compiler generates an executable named `a.out` on UNIX systems and `a.exe` on Windows. (Note: Depending on the release of the GNU compiler you are using, you may need to specify `-std=c++0x` to turn on C++ 11 support.)

The command to run the Microsoft Visual Studio 2010 compiler is `cl`:

Click here to view code image

```
C:\Users\me\Programs> cl /EHsc prog1.cpp
```

Here `C:\Users\me\Programs>` is the system prompt and `\Users\me\Programs` is the name of the current directory (aka the current folder). The `cl` command invokes the compiler, and `/EHsc` is the compiler option that turns on standard exception handling. The Microsoft compiler automatically generates an executable with a name that corresponds to the first 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`.

Compilers usually include options to generate warnings about problematic constructs. It is usually a good idea to use these options. Our preference is to use `-Wall` with the GNU compiler, and to use `/W4` with the Microsoft compilers.

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

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. Recompile and rerun your program to see how your system treats a failure indicator from `main`.

1.2. A First Look at Input/Output

The C++ language does not define any statements to do input or output (IO). Instead, C++ includes an extensive **standard library** that provides IO (and many other facilities). For many purposes, including the examples in this book, one needs to know only a few basic concepts and operations from the IO library.

Most of the examples in this book use the **iostream** library. 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 read from or written to an IO device. The term *stream* is intended to suggest that the characters are generated, or consumed, sequentially over time.

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*). This object is also known as the **standard output**. The library also defines two other `ostream` objects, named **cerr** and **clog** (pronounced *see-err* and *see-log*, respectively). We typically use `cerr`, referred to as the **standard error**, for warning and error messages and `clog` 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 are 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.

A Program That Uses the IO Library

In our bookstore problem, we'll have several records that we'll want to combine into a single total. As a simpler, related problem, let's look first at how we might add two numbers. Using the IO library, we can extend our `main` program to prompt the user to give us two numbers and then print their sum:

Click here to view code image

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;
    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

```
#include <iostream>
```

tells the compiler that we want to use the `iostream` library. The name inside angle brackets (`iostream` in this case) refers to 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 must appear outside any function. Typically, we put all the `#include` directives for a program at the beginning of the source file.

Writing to a Stream

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

Click here to view code image

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

The << 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 the given value on the given `ostream`. The result of the output operator is its left-hand operand. That is, the result is the `ostream` on which we wrote the given value.

Our output statement uses the << operator twice. Because the operator returns its left-hand operand, the result of the first operator becomes the left-hand operand of the second. As a result, we can chain together output requests. Thus, our expression is equivalent to

Click here to view code image

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

Each operator in the chain has the same object as its left-hand operand, in this case `std::cout`. Alternatively, we can generate the same output using two statements:

Click here to view code image

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

The first output operator prints a message to the user. That message is a **string**

literal, which is a sequence of characters enclosed in double quotation marks. The text between the quotation marks is printed to the standard output.

The second operator prints `endl`, which is a special value called a **manipulator**. Writing `endl` has the effect of ending the current line and flushing the *buffer* associated with that device. Flushing the buffer ensures that all the output the program has generated so far is actually written to the output stream, rather than sitting in memory waiting to be written.

> **Warning**
>
> Programmers often add print statements during debugging. Such statements should *always* flush the stream. Otherwise, if the program crashes, output may be left in the buffer, 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 us to avoid inadvertent collisions between the names we define and uses of those same names inside a library. All the names defined by the standard library are in the `std` namespace.

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`. § 3.1 (p. 82) will show a simpler way to access names from the library.

Reading from a Stream

Having asked the user for input, we next want to read that input. We start by defining two *variables* named `v1` and `v2` to hold the input:

```
int v1 = 0, v2 = 0;
```

We define these variables as type `int`, which is a built-in type representing integers. We also *initialize* them to `0`. When we initialize a variable, we give it the indicated value at the same time as the variable is created.

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 data from the given `istream` and stores what was read in the given object. Like the output operator, the input operator returns its left-hand operand as its result. Hence, this expression is equivalent to

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

Because the operator returns its left-hand operand, we can combine a sequence of input requests into a single statement. Our input operation reads two values from `std::cin`, storing the first in `v1` and the second in `v2`. In other words, our input operation executes as

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

Completing the Program

What remains is to print our result:

Click here to view code image

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

This statement, although longer than the one that prompted the user for input, is conceptually similar. It prints each of its operands on the standard output. What is interesting in this example is that the operands are not all the same kinds of values. Some operands are string literals, such as `"The sum of "`. Others are `int` values, such as `v1`, `v2`, and the result of evaluating the arithmetic expression `v1 + v2`. The library defines versions of the input and output operators that handle operands of each of these differing types.

Exercises Section 1.2

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

Exercise 1.4: Our program used the addition operator, `+`, to add two numbers. Write a program that uses the multiplication operator, `*`, to print the product instead.

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 whether the following program fragment is legal.

Click here to view code image

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

If the program is legal, what does it do? If the program is not legal, why not? How would you fix it?

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. The compiler ignores comments, so they have no effect on the program's behavior or performance.

Although the compiler ignores comments, readers of our code do not. Programmers tend to believe comments even when other parts of the system documentation are out of date. An incorrect comment is worse than no comment at all because it may mislead the reader. When you change your code, be sure to update the comments, too!

Kinds of Comments in C++

There are two kinds of comments in C++: single-line and paired. A single-line comment starts with a double slash (`//`) and ends with a newline. Everything to the right of the slashes on the current line is ignored by the compiler. A comment of this kind can contain any text, including additional double slashes.

The other kind of comment uses two delimiters (`/*` and `*/`) that are inherited from C. Such comments begin with a `/*` and end with the next `*/`. These comments can include anything that is not a `*/`, including newlines. The compiler treats everything that falls between the `/*` and `*/` as part of the comment.

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 multiline comment. Our style is to begin each line in the comment with an asterisk, thus indicating that the entire range is part of a multiline comment.

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

Click here to view code image

```
#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 = 0, v2 = 0;    //  variables to hold the input we read
    std::cin >> v1 >> v2; //  read input
    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1 + v2 << std::endl;
    return 0;
}
```

> **Note**
>
> 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 you are using.

Comment Pairs Do Not Nest

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

Click here to view code image

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

We often need to comment out a block of code during debugging. Because that code might contain nested comment pairs, the best way to comment a block of code is to insert single-line comments at the beginning of each line in the section we want to ignore:

Click here to view code image

```
// /*
```

```
// * everything inside a single-line comment is ignored
// * including nested comment pairs
//   */
```

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:

Click here to view code image

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

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

1.4. Flow of Control

Statements normally execute sequentially: The first statement in a block is executed first, followed by the second, and so on. Of course, few programs—including the one to solve our bookstore problem—can be written using only sequential execution. Instead, programming languages provide various flow-of-control statements that allow for more complicated execution paths.

1.4.1. The while Statement

A **while statement** repeatedly executes a section of code so long as a given condition is true. We can use a `while` to write a program to sum the numbers from 1 through 10 inclusive as follows:

Click here to view code image

```
#include <iostream>
int main()
{
    int sum = 0, val  = 1;
    // keep executing the while as long as val is less than or equal to 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;
}
```

When we compile and execute this program, it prints

Sum of 1 to 10 inclusive is 55

As before, we start by including the `iostream` header and defining `main`. 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 `0` and start `val` off with the value `1`.

The new part of this program is the `while` statement. A `while` has the form

```
while (condition)
    statement
```

A `while` executes by (alternately) testing the *condition* and executing the associated *statement* until the *condition* is false. A **condition** is an expression that yields a result that is either true or false. So long as *condition* is true, *statement* is executed. After executing *statement*, *condition* is tested again. If *condition* is again true, then *statement* is again executed. The `while` continues, alternately testing the *condition* and executing *statement* until the *condition* is false.

In this program, the `while` statement is

Click here to view code image

```
// keep executing the while as long as val is less than or equal to 10
while (val <= 10) {
    sum += val;    // assigns sum + val to sum
    ++val;         // add 1 to val
}
```

The condition 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, the condition is true. If the condition is true, we execute the body of the `while`. In this case, that body is a block with two statements:

Click here to view code image

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

A block is a sequence of zero or more statements enclosed by curly braces. A block is a statement and may be used wherever a statement is required. The first statement in this block uses the compound assignment operator (the **+= operator**). This operator adds its right-hand operand to its left-hand operand and stores the result in the left-hand operand. It has essentially the same effect as writing an addition and an

assignment:

Click here to view code image

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

Thus, the first statement in the block 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 `1` to its operand. Writing `++val` is the same as writing `val = val + 1`.

After executing the `while` body, the loop evaluates the condition again. 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, the program falls out of the `while` loop and continues execution with the statement following the `while`. In this case, that statement prints our output, followed by the `return`, which completes our `main` program.

Exercises Section 1.4.1

Exercise 1.9: Write a program that uses a `while` to sum the numbers from 50 to 100.

Exercise 1.10: In addition to the `++` operator that adds `1` to its operand, there is a decrement operator (`--`) that subtracts `1`. Use the decrement operator to write a `while` that prints the numbers from ten down to zero.

Exercise 1.11: Write a program that prompts the user for two integers. Print each number in the range specified by those two integers.

1.4.2. The for Statement

In our `while` loop we used the variable `val` to control how many times we executed the loop. We tested the value of `val` in the condition and incremented `val` in the `while` body.

This pattern—using a variable in a condition and incrementing that variable in the body—happens so often that the language defines a second statement, the **for statement**, that abbreviates code that follows this pattern. We can rewrite this program using a `for` loop to sum the numbers from 1 through 10 as follows:

Click here to view code image

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 through 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;
}
```

As before, we define `sum` and initialize it to zero. In this version, we define `val` as part of the `for` statement itself:

Click here to view code image

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

Each `for` statement has two parts: a header and a 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 `1`. The variable `val` exists only inside the `for`; it is not possible to use `val` after this loop terminates. The *init-statement* is executed only once, on entry to the `for`. The *condition*

```
val <= 10
```

compares the current value in `val` to `10`. The *condition* is tested each time through the loop. As long as `val` is less than or equal to `10`, we execute the `for` body. The *expression* is executed after the `for` body. Here, the *expression*

```
++val
```

uses the prefix increment operator, which adds `1` 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 again. After executing the body, `val` is incremented again. The loop continues until the *condition* fails.

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

Click here to view code image

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

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`. If the test succeeds, execute the `for` body. If the test fails, exit the loop and continue execution with the first statement following the `for` body.

3. Increment `val`.

4. Repeat the test in step 2, continuing with the remaining steps as long as the condition is true.

Exercises Section 1.4.2

Exercise 1.12: What does the following `for` loop do? What is the final value of `sum`?

Click here to view code image

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

Exercise 1.13: Rewrite the exercises from § 1.4.1 (p. 13) using `for` loops.

Exercise 1.14: Compare and contrast the loops that used a `for` with those using a `while`. Are there advantages or disadvantages to using either form?

Exercise 1.15: Write programs that contain the common errors discussed in the box on page 16. Familiarize yourself with the messages the compiler generates.

1.4.3. Reading an Unknown Number of Inputs

In the preceding sections, we wrote programs that summed the numbers from 1 through 10. A logical extension of this program would be to ask the user to input a set of numbers to sum. In this case, we won't know how many numbers to add. Instead, we'll keep reading numbers until there are no more numbers to read:

Click here to view code image

```
#include <iostream>
int main()
{
    int sum = 0, value = 0;
    // read until 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

The first line inside `main` defines two `int` variables, named `sum` and `value`, which we initialize to `0`. We'll use `value` to hold each number as we read it from the input. We read the data inside the condition of the `while`:

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

Evaluating the `while` condition executes the expression

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

That expression reads the next number from the standard input and stores that number in `value`. The input operator (§ 1.2, p. 8) returns its left operand, which in this case is `std::cin`. This condition, therefore, 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 the stream hasn't encountered an error—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 yield false.

Thus, our `while` executes until we encounter end-of-file (or an input error). The `while` body uses the compound assignment operator to add the current value to the evolving `sum`. Once the condition fails, the `while` ends. We fall through and execute the next statement, which prints the `sum` followed by `endl`.

Entering an End-of-File from the Keyboard

When we enter input to a program from the keyboard, different operating systems use different conventions to allow us to indicate end-of-file. On Windows systems we enter an end-of-file by typing a control-z—hold down the Ctrl key and press z—followed by hitting either the Enter or Return key. On UNIX systems, including on Mac OS X machines, end-of-file is usually control-d.

Compilation Revisited

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

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:

Click here to view code image

```
// 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;
    // error: second output operator is missing
    std::cout << "Write new master." std::endl;
    // error: missing ; on return statement
    return 0
}
```

Type errors: Each item of data in C++ has an associated type. The value 10, for example, has a type of `int` (or, more colloquially, "is an `int`"). The word `"hello"`, including the double quotation marks, is a string literal. One example of a type error is passing a string literal to a function that expects an `int` argument.

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 forgetting to use `std::` for a name from the library and misspelling the name of an identifier:

Click here to view code image

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

Error messages usually contain 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 1.4.3

Exercise 1.16: Write your own version of a program that prints the sum of a set of integers read from `cin`.

1.4.4. The if Statement

Like most languages, C++ provides an **if statement** that supports conditional execution. We can use an `if` to write a program to count how many consecutive times each distinct value appears in the input:

Click here to view code image

```
#include <iostream>
int main()
{
    // currVal is the number we're counting; we'll read new values into val
    int currVal = 0, val = 0;
    // read first number and ensure that we have data to process
    if (std::cin >> currVal) {
        int cnt = 1;  // store the count for the current value we're processing
        while (std::cin >> val) { // read the remaining numbers
            if (val == currVal)   // if the values are the same
                ++cnt;            // add 1 to cnt
            else { // otherwise, print the count for the previous value
                std::cout << currVal << " occurs "
                          << cnt << " times" << std::endl;
                currVal = val;    // remember the new value
                cnt = 1;          // reset the counter
            }
        }  // while loop ends here
        // remember to print the count for the last value in the file
        std::cout << currVal <<  " occurs "
                  << cnt << " times" << std::endl;
    } // outermost if statement ends here
    return 0;
}
```

If we give this program the following input:

Click here to view code image

```
42 42 42 42 42 55 55 62 100 100 100
```

then the output should be

```
42 occurs 5 times
55 occurs 2 times
62 occurs 1 times
100 occurs 3 times
```

Much of the code in this program should be familiar from our earlier programs. We start by defining `val` and `currVal`: `currVal` will keep track of which number we are counting; `val` will hold each number as we read it from the input. What's new are the two `if` statements. The first `if`

Click here to view code image

```
if (std::cin >> currVal) {
    // ...
} // outermost if statement ends here
```

ensures that the input is not empty. Like a `while`, an `if` evaluates a condition. The condition in the first `if` reads a value into `currVal`. If the read succeeds, then the condition is true and we execute the block that starts with the open curly following the condition. That block ends with the close curly just before the `return` statement.

Once we know there are numbers to count, we define `cnt`, which will count how often each distinct number occurs. We use a `while` loop similar to the one in the previous section to (repeatedly) read numbers from the standard input.

The body of the `while` is a block that contains the second `if` statement:

Click here to view code image

```
if (val == currVal)    // if the values are the same
    ++cnt;             // add 1 to cnt
else { // otherwise, print the count for the previous value
    std::cout << currVal << " occurs "
              << cnt << " times" << std::endl;
    currVal = val;     // remember the new value
    cnt = 1;           // reset the counter
}
```

The condition in this `if` uses the equality operator (the **== operator**) to test whether `val` is equal to `currVal`. If so, we execute the statement that immediately follows the condition. That statement increments `cnt`, indicating that we have seen `currVal` once more.

If the condition is false—that is, if `val` is not equal to `currVal`—then we execute the statement following the `else`. This statement is a block consisting of an output statement and two assignments. The output statement prints the count for the value we just finished processing. The assignments reset `cnt` to `1` and `currVal` to `val`, which is the number we just read.

Warning

C++ uses = for assignment and == for equality. Both operators can appear inside a condition. It is a common mistake to write = when you mean == inside a condition.

Exercises Section 1.4.4

Exercise 1.17: What happens in the program presented in this section if the input values are all equal? What if there are no duplicated values?

Exercise 1.18: Compile and run the program from this section giving it only equal values as input. Run it again giving it values in which no number is repeated.

Exercise 1.19: Revise the program you wrote for the exercises in § 1.4.1 (p. 13) that printed a range of numbers so that it handles input in which the first number is smaller than the second.

Key Concept: Indentation and Formatting of C++ Programs

C++ programs are largely free-format, meaning that where we put curly braces, indentation, comments, and newlines usually has no effect on what our programs mean. 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 else we'd like. The only requirement is that the open curly must be the first nonblank, noncomment character following `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. Most programmers indent subsidiary parts of their programs, as we've done with the statements inside `main` and the bodies of our loops. We tend to put the curly braces that delimit functions on their own lines. We also indent compound IO expressions so that the operators line up. Other indentation conventions will become clear as our programs become more sophisticated.

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

1.5. Introducing Classes

The only remaining feature we need to understand before solving our bookstore problem is how to define a *data structure* to represent our transaction data. In C++ we define our own data structures by defining a **class**. A class defines a type along with a collection of operations that are related to that type. 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.

In this section, we'll describe a simple class that we can use in writing our bookstore program. We'll implement this class in later chapters as we learn more about types, expressions, statements, and functions.

To use a class we need to know three things:

- What is its name?
- Where is it defined?
- What operations does it support?

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

As we've seen, to use a library facility, we must include the associated header. Similarly, we use headers to access classes defined for our own applications. Conventionally, header file names are derived from the name of a class defined in that header. Header files that we write usually have a suffix of `.h`, but some programmers use `.H`, `.hpp`, or `.hxx`. The standard library headers typically have no suffix at all. Compilers usually don't care about the form of header file names, but IDEs sometimes do.

1.5.1. The Sales_item Class

The purpose of the `Sales_item` class is to represent the total revenue, number of copies sold, and average sales price for a book. How these data are stored or computed is not our concern. To use a class, we need not care about how it is implemented. Instead, what we need to know is what operations objects of that type can perform.

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:

• Call a function named `isbn` to fetch the from a `Sales_item` object.
• Use the input (`>>`) and output (`<<`) operators to read and write objects of type `Sales_item`.
• Use the assignment operator (`=`) to assign one `Sales_item` object to another.
• Use the addition operator (`+`) to add two `Sales_item` objects. The two objects must refer to the same ISBN. The result is a new `Sales_item` object whose ISBN is that of its operands and whose number sold and revenue are the sum of the corresponding values in its operands.
• Use the compound assignment operator (`+=`) to add one `Sales_item` object into another.

Key Concept: Classes Define Behavior

The important thing to keep in mind when you read these programs is that the author of the `Sales_item` class defines *all* the actions that can be performed by objects of this class. That is, the `Sales_item` class defines what happens when a `Sales_item` object is created and what happens when the assignment, addition, or the input and output operators are applied to `Sales_items`.

In general, the class author determines all the operations that can be used on objects of the class type. For now, the only operations we know we can perform on `Sales_item` objects are the ones listed in this section.

Reading and Writing Sales_items

Now that we know what operations we can use with `Sales_item` objects, we can write programs that use the class. For example, the following program reads data from the standard input into a `Sales_item` object and writes that `Sales_item` back onto the standard output:

Click here to view code image

```
#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 says 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. Headers from the standard library are enclosed in angle brackets (`< >`). Those that are not part of the library are enclosed in double quotes (`" "`).

Inside `main` we define an object, named `book`, that 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 by printing `endl`.

Adding Sales_items

A more interesting example adds two `Sales_item` objects:

Click here to view code image

```
#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 transactions. We read data into these objects from the standard input. The output expression does the addition and prints the result.

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 this similarity noteworthy 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.

Using File Redirection

It can be tedious to repeatedly type these transactions as input to the programs you are testing. Most operating systems support file redirection, which lets us associate a named file with the standard input and the standard output:

```
$ addItems <infile >outfile
```

Assuming $ is the system prompt and our addition program has been compiled into an executable file named `addItems.exe` (or `addItems` on UNIX systems), this command will read transactions from a file named `infile` and write its output to a file named `outfile` in the current directory.

Exercises Section 1.5.1

Exercise 1.20: http://www.informit.com/title/032174113 contains a copy of `Sales_item.h` in the Chapter 1 code directory. Copy that file to your working directory. Use it to write a program that reads a set of book sales transactions, writing each transaction to the standard output.

Exercise 1.21: Write a program that reads two `Sales_item` objects that have the same ISBN and produces their sum.

Exercise 1.22: 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

Our program that adds two `Sales_items` should check whether the objects have the same ISBN. We’ll do so as follows:

Click here to view code image

```
#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.isbn() == item2.isbn()) {
            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 version is the `if` and its associated `else` branch. Even without understanding the `if` condition, we know what this program does. If the condition succeeds, then we write the same output as before and return `0`, indicating success. If the condition 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

```
item1.isbn() == item2.isbn()
```

calls a **member function** named `isbn`. A member function is a function that is defined as part of a class. Member functions are sometimes referred to as **methods**.

Ordinarily, we call a member function on behalf of an object. For example, the first part of the left-hand operand of the equality expression

```
item1.isbn
```

uses the dot operator (the **"." operator**) to say that we want "the `isbn` member of the object named `item1`." The dot operator applies only to objects of class type. The left-hand operand must be an object of class type, and the right-hand operand must name a member of that type. The result of the dot operator is the member named by the right-hand operand.

When we use the dot operator to access a member function, we usually do so to call that function. We call a function using the call operator (the **() operator**). The call operator is a pair of parentheses that enclose a (possibly empty) list of *arguments*. The `isbn` member function does not take an argument. Thus,

```
item1.isbn()
```

calls the `isbn` function that is a member of the object named `item1`. This function returns the ISBN stored in `item1`.

The right-hand operand of the equality operator executes in the same way—it returns the ISBN stored in `item2`. If the ISBNs are the same, the condition is true; otherwise it is false.

Exercises Section 1.5.2

Exercise 1.23: Write a program that reads several transactions and counts how many transactions occur for each ISBN.

Exercise 1.24: Test the previous program by giving multiple transactions representing multiple ISBNS. The records for each ISBN should be grouped together.

1.6. The Bookstore Program

We are now 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 number of copies sold, the total revenue, and the average sales price. We'll assume that all the transactions for each ISBN are grouped together in the input.

Our program will combine the data for each ISBN in a variable named `total`. We'll use a second variable named `trans` to hold each transaction we read. If `trans` and `total` refer to the same ISBN, we'll update `total`. Otherwise we'll print `total` and reset it using the transaction we just read:

Click here to view code image

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item total; // variable to hold data for the next transaction
    // read the first transaction and ensure that there are data to process
    if (std::cin >> total) {
        Sales_item trans; // variable to hold the running sum
        // read and process the remaining transactions
        while (std::cin >> trans) {
            // if we're still processing the same book
            if (total.isbn() == trans.isbn())
                total += trans; // update the running total
            else {
                // print results for the previous book
                std::cout << total << std::endl;
                total = trans;  // total now refers to the next book
            }
        }
        std::cout << total << std::endl; // print the last transaction
    } else {
        // no input! warn the user
        std::cerr << "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 seen.

As usual, we begin by including the headers that we use, `iostream` from the library and our own `Sales_item.h`. Inside `main` we define an object named `total`, which we’ll use to sum the data for a given ISBN. We start by reading the first 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 tells the user that there was no input.

Assuming we have successfully read a record, we execute the block following the outermost `if`. That block starts by defining the object named `trans`, which will hold our transactions as we read them. The `while` statement will read all the remaining records. As in our earlier programs, the `while` condition reads a value from the standard input. 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. The `if` checks whether the ISBNS are equal. If so, we use the compound assignment operator to add `trans` to `total`. If the ISBNS are not equal, we print the value stored in `total` and reset `total` by assigning `trans` to it. After executing the `if`, we return to the condition in the `while`, reading the next transaction, and so on until we run out of records.

When the `while` terminates, `total` contains the data for the last ISBN in the file. We write the data for the last ISBN 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.

Chapter Summary

This chapter introduced enough of C++ to let you compile and execute simple C++ programs. We saw how to define a `main` function, which is the function that the operating system calls to execute our 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 class that someone else has defined. Later chapters will show how to define our own classes.

Defined Terms

argument Value passed to a function.

assignment Obliterates an object's current value and replaces that value by a new one.

block Sequence of zero or more 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 from actions in the program. Output buffers can 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 Type, such as `int`, defined by the language.

cerr `ostream` object tied to the standard error, which often writes to the same device 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.

character string literal Another term for string literal.

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

class Facility for defining our own data structures together with associated operations. 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. A value of 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 that indicates that there is no more input in a file.

expression The smallest unit of computation. An expression consists of one or more operands and usually one or more operators. Expressions are evaluated to produce a result. For example, assuming `i` and `j` are `int`s, then `i + j` is an expression and yields the sum of the two `int` values.

for statement Iteration statement that provides iterative execution. Often used to repeat a calculation a fixed number of times.

function Named unit of computation.

function body Block that defines the actions performed by a function.

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

header Mechanism whereby the definitions of a class or other names are made available to multiple programs. A program uses a header 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, the `else` body is executed if there is one.

initialize Give an object a value at the same time that it is created.

iostream Header that provides the library types for stream-oriented input and output.

istream Library type providing stream-oriented input.

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

main Function called by the operating system to execute 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.

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.

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 Output stream used for error reporting. Ordinarily, the standard output and the standard error are tied to the window in which the program is executed.

standard input Input stream usually associated with the window in which the program executes.

standard library Collection of types and functions that every C++ compiler must support. The library provides the types that support IO. C++ programmers tend to talk about "the library," meaning the entire standard library. They also tend to refer to particular parts of the library by referring to a library type, such as the "`iostream` library," meaning the part of the standard library that defines the IO classes.

standard output Output stream usually associated with the window in which the program executes.

statement A part of a program that specifies an action to take place when the program is executed. An expression followed by a semicolon is a statement; other kinds of statements include blocks and `if`, `for`, and `while` statements, all of which contain other statements within themselves.

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 zero or more characters enclosed in double quotes (`"a string literal"`).

uninitialized variable Variable that is not given an initial value. Variables of class type for which no initial value is specified are initialized as specified by the class definition. Variables of built-in type defined inside a function are uninitialized unless explicitly initialized. It is an error to try to use the value of an uninitialized variable. *Uninitialized variables are a rich source of bugs*.

variable A named object.

while statement Iteration statement that provides iterative execution so long as a specified condition is true. The body is executed zero or more times, depending on the truth value of the condition.

() operator 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 `1` to the operand; `++i` is equivalent to `i = i + 1`.

+= operator Compound assignment operator that adds the right-hand operand to the left and stores the result in the left-hand operand; `a += b` is equivalent to `a = a + b`.

. operator Dot operator. Left-hand operand must be an object of class type and the right-hand operand must be the name of a member of that object. The operator yields the named member of the given object.

:: operator Scope operator. Among other uses, the scope operator is used to access names in a namespace. For example, `std::cout` denotes 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 Decrement operator. Subtracts `1` from the operand; `--i` is equivalent to `i = i - 1`.

<< 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`.

 # include Directive that makes code in a header available to a program.

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

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

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

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

>= operator Greater-than-or-equal operator. Tests whether the left-hand