KENT C++ DEMYSTIFIED

QA 76
.73
.C153
K457
2004
Copy 2

C++ DeMYSTiFieD

A SELF-TEACHING GUIDE

No formal training in C++ needed!

Create and run your own computer programs

Many examples illustrating application of concepts

Complete with chapter-ending quizzes and final exam

Jeff Kent

McGraw Hill Osborne

C++ DEMYSTIFIED

JEFF KENT

McGraw-Hill/Osborne

New York Chicago San Francisco Lisbon London
Madrid Mexico City Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

The McGraw·Hill Companies

McGraw-Hill/Osborne
2100 Powell Street, 10th Floor
Emeryville, California 94608
U.S.A.

To arrange bulk purchase discounts for sales promotions, premiums, or fund-raisers, please contact **McGraw-Hill**/Osborne at the above address. For information on translations or book distributors outside the U.S.A., please see the International Contact Information page immediately following the index of this book.

C++ Demystified

1234567890 FGR FGR 01987654

ISBN 0-07-225370-3

QA76
.73
.C153K457
2004
Copy 2

Publisher
Brandon A. Nordin

Vice President & Associate Publisher
Scott Rogers

Editorial Director
Wendy Rinaldi

Project Editor
Lisa Wolters-Broder

Acquisitions Coordinator
Athena Honore

Technical Editor
Jim Keogh

Copy Editor
Mike McGee

Proofreader
Susie Elkind

Indexer
Irv Hershman

Composition
Apollo Publishing Services, Lucie Ericksen

Illustrators
Kathleen Edwards, Melinda Lytle

Cover Series Design
Margaret Webster-Shapiro

Cover Illustration
Lance Lekander

This book was composed with Corel VENTURA™ Publisher.

2004351484

CONTENTS AT A GLANCE

ABOUT THE AUTHOR

Jeff Kent is an Associate Professor of Computer Science at Los Angeles Valley College in Valley Glen, California. He teaches a number of programming languages, including Visual Basic, C++, Java and, when he's feeling masochistic, Assembler, but mostly he teaches C++. He also manages a network for a Los Angeles law firm whose employees are guinea pigs for his applications, and as an attorney gives advice to young attorneys whether they want it or not. He also has written several books on computer programming, including the recent *Visual Basic.NET A Beginner's Guide* for McGraw-Hill/Osborne.

Jeff has had a varied career—or careers. He graduated from UCLA with a Bachelor of Science degree in economics, then obtained a Juris Doctor degree from Loyola (Los Angeles) School of Law, and went on to practice law. During this time, when personal computers still were a gleam in Bill Gates's eye, Jeff was also a professional chess master, earning a third-place finish in the United States Under-21 Championship and, later, an international title.

Jeff does find time to spend with his wife, Devvie, which is not difficult since she also is a computer science professor at Valley College. He also acts as personal chauffeur for his teenaged daughter, Emily (his older daughter, Elise, now has her own driver's license) and in his remaining spare time enjoys watching international chess tournaments on the Internet. His goal is to resume running marathons, since otherwise, given his losing battle to lose weight, his next book may be *Sumo Wrestling Demystified.*

I would like to dedicate this book to my wife, Devvie Schneider Kent. There is not room here to describe how she has helped me in my personal and professional life, though I do mention several ways in the Acknowledgments. She also has been my computer programming teacher in more ways than one; I wouldn't be writing this and other computer programming books if it wasn't for her.

—Jeff Kent

CONTENTS

ACKNOWLEDGMENTS

It seems obligatory in acknowledgments for authors to thank their publishers (especially if they want to write for them again), but I really mean it. This is my fourth book for McGraw-Hill/Osborne, and I hope there will be many more. It truly is a pleasure to work with professionals who are nice people as well as very good at what they do (even when what they are good at is keeping accurate track of the deadlines I miss).

I first want to thank Wendy Rinaldi, who got me started with McGraw-Hill/Osborne back in 1998 (has it been that long?). Wendy was also my first Acquisitions Editor. Indeed, I got started on this book through a telephone call with Wendy at the end of a vacation with my wife, Devvie, who, being in earshot, and with an "are you insane" tone in her voice, asked incredulously, "You're writing another book?"

I also must thank my Acquisitions Coordinator, Athena Honore, and my Project Editor, Lisa Wolters-Broder. Both were unfailingly helpful and patient, while still keeping me on track in this deadline-sensitive business (e.g., "I'm so sorry you broke both your arms and legs; you'll still have the next chapter turned in by this Friday, right?").

Mike McGee did the copyediting, together with Lisa. They were kind about my obvious failure during my school days to pay attention to my grammar lessons. They improved what I wrote while still keeping it in my words (that way, if something is wrong, it is still my fault). Mike also indicated he liked some of my stale jokes, which makes him a friend for life.

Jim Keogh was my technical editor. Jim and I had a balance of terror going between us, in that while he was tech editing this book, I was tech editing two books on which he was the main author, *Data Structures Demystified* and *OOP Demystified*. Seriously, Jim's suggestions were quite helpful and added value to this book.

There are a lot of other talented people behind the scenes who also helped get this book out to press, but, as in an Academy Awards speech, I can't list them all. That doesn't mean I don't appreciate all their hard work, because I do.

I truly thank my wife Devvie, who in addition to being my wife, best friend (maybe my only one), and partner (I'm leaving out lover because computer programmers aren't supposed to be interested in such things), also was my personal tech editor. She is well-qualified for that task, since she has been a computer science professor for 15 years, and also is a stickler for correct English (yes, I know, you can't modify the word "unique"). She made this a much better book.

Finally, I would like to give thanks to my daughters, Elise and Emily, and my mom, Bea Kent, for tolerating me when I excused myself from family gatherings, muttering to myself about unreasonable chapter deadlines and merciless editors (sorry, Athena and Lisa). I also should thank my family in advance for not having me committed when I talk about writing my next book.

INTRODUCTION

C++ was my first programming language. While I've since learned others, I've always thought C++ was the "best" programming language, perhaps because of the power it gives the programmer. Of course, this power is a double-edged sword, being also the power to hang yourself if you are not careful. Nonetheless, C++ has always been my favorite programming language.

C++ also has been the first choice of others, not just in the business world because of its power, but also in academia. Additionally, many other programming languages, including Java and C#, are based on C++. Indeed, the Java programming language was written using C++. Therefore, knowing C++ also makes learning other programming languages easier.

Why Did I Write this Book?

Not as a road to riches, fame, or beautiful women. I may be misguided, but I'm not completely delusional.

To be sure, there are many introductory level books on C++. Nevertheless, I wrote this book because I believe I bring a different and, I hope, valuable perspective.

As you may know from my author biography, I teach computer science at Los Angeles Valley College, a community college in the San Fernando Valley area of Los Angeles, where I grew up and have lived most of my life. I also write computer programs, but teaching programming has provided me with insights into how students learn that I could never obtain from writing programs. These insights are gained not just from answering student questions during lectures. I spend hours each week in our college's computer lab helping students with their programs, and more hours each week reviewing and grading their assignments. Patterns emerge regarding which teaching methods work and which don't, the order in which to introduce programming topics, the level of difficulty at which to introduce a new topic, and so on. I joke with my students that they are my beta testers in my never-ending attempt to become a better teacher, but there is much truth in that joke.

Additionally, my beta testers… err, students, seem to complain about the textbook no matter which book I adopt. Many ask me why I don't write a book they could use to learn C++. They may be saying this to flatter me (I'm not saying it doesn't work), or for the more sinister reason that they will be able to blame the teacher for a poor book as well as poor instruction. Nevertheless, having written other books, these questions planted in my mind the idea of writing a book that, in addition to being sold to the general public, also could be used as a supplement to a textbook.

Who Should Read this Book

Anyone who will pay for it! Just kidding, though no buyers will be turned away.

It is hardly news that publishers and authors want the largest possible audience for their books. Therefore, this section of the introduction usually tells you this book is for you whoever you may be and whatever you do. However, no programming book is for everyone. For example, if you exclusively create game programs using Java, this book may not be for you (though being a community college teacher I may be your next customer if you create a space beasts vs. community college administrators game).

While this book is, of course, not for everyone, it very well may be for you. Many people need or want to learn C++, either as part of a degree program, job training, or even as a hobby. C++ is not the easiest subject to learn, and unfortunately many books don't make learning C++ any easier, throwing at you a veritable telephone book of complexity and jargon. By contrast, this book, as its title suggests, is designed to "demystify" C++. Therefore, it goes straight to the core concepts and explains them in a logical order and in plain English.

What this Book Covers

I strongly believe that the best way to learn programming is to write programs. The concepts covered by the chapters are illustrated by clearly and thoroughly explained code. You can run this code yourself, or use the code as the basis for writing further programs that expand on the covered concepts.

Chapter 1 gets you started. This chapter answers questions such as what is a computer program and what is a programming language. It then discusses the anatomy of a basic C++ program, including both the code you see and what happens "under the hood," explaining how the preprocessor, compiler, and linker work together to translate your code into instructions the computer can understand. Finally, the chapter tells you how to use an integrated development environment (IDE) to create and run a project.

Being able to create and run a program that outputs "Hello World!" as in Chapter 1 is a good start. However, most programs require the storing of information of different types, such as numeric and text. Chapter 2 first explains the different types of computer memory, including random access memory, or RAM. The chapter then discusses addresses, which identify where data is stored in RAM, and bytes, the unit of value for the amount of space required to store information. Because information comes in different forms, this chapter next discusses the different data types for whole numbers, floating point numbers and text.

The featured star of Chapter 3 is the variable, which not only reserves the amount of memory necessary to store information, but also provides you with a name by which that information later may be retrieved. Because the purpose of a variable is to store a value, a variable without an assigned value is as pointless as a bank account without money. Therefore, this chapter explains how to assign a value to a variable, either at compile time using the assignment operator or at run time using the cin object and the stream extraction operator.

As a former professional chess player, I have marveled at the ability of chess computers to play world champions on even terms. The reason the chess computers have this ability is because they can calculate far more quickly and accurately than we can. Chapter 4 covers arithmetic operators, which we use in code to harness the computer's calculating ability.

As programs become more sophisticated, they often branch in two or more directions based on whether a condition is true or false. For example, while a calculator program would use the arithmetic operators you learned about in Chapter 4, your program first would need to determine whether the user chose addition, subtraction, multiplication, or division before performing the indicated arithmetic operation. Chapters 5 and 6 introduce relational and logical operators, which are useful in determining a user's choice, and the if and switch statements, used to direct the path the code will follow based on the user's choice.

When you were a child, your parents may have told you not to repeat yourself. However, sometimes your code needs to repeat itself. For example, if an application user enters invalid data, your code may continue to ask the user whether they want to retry or quit until the user either enters valid data or quits. The primary subject of Chapters 7 and 8 are loops, which are used to repeat code execution until a condition is no longer true. Chapter 7 starts with the for loop, and also introduces the increment and decrement operators, which are very useful when working with loops. Chapter 8 completes the discussion of loops with the while and do while loops.

Chapter 9 is about functions. A function is a block of one or more code statements. All of your C++ code that executes is written within functions. This chapter will explain why and how you should write your own functions. It first explains how to prototype and define a function, and then how to call the function. This chapter also explains how you use arguments to pass information from the calling function to a called function and a return value to pass information back from the called function to a calling function. Passing by value and by reference also are explained and distinguished. This chapter winds up explaining variable scope and lifetime, and both explaining and distinguishing local, static, and global variables.

Chapter 10 is about arrays. Unlike the variables covered previously in the book, which may hold only one value at a time, arrays may hold multiple values at one time. Additionally, arrays work very well with loops, which are covered in Chapters 7 and 8. This chapter also distinguishes character arrays from arrays of other data types. Finally, this chapter covers constants, which are similar to variables, but differ in that their initial value never changes while the program is running.

Chapter 11 is about pointers. The term pointers often strikes fear in the heart of a C++ student, but it shouldn't. As you learned back in Chapters 2 and 3, information is stored at addresses in memory. Pointers simply provide you with an efficient way to access those addresses. You also will learn in this chapter about the indirection operator and dereferencing as well as pointer arithmetic.

Most information, including user input, is in the form of character, C-string, and C++ string class data types. Chapter 12 shows you functions that are useful in working with these data types, including member functions of the cin object.

Information is stored in files so it will be available after the program ends. Chapter 13 teaches you about the file stream objects,*fstream*, *ifstream,* and *ofstream*, and how to use them and their member functions to open, read, write and close files.

Finally, to provide you with a strong basis to go to the next step after this introductory level book, Chapter 14 introduces you to OOP, Object-Oriented Programming, and two programming concepts heavily used in OOP, structures and classes.

A Quiz follows each chapter. Each quiz helps you confirm that you have absorbed the basics of the chapter. Unlike quizzes you took in school, you also have an answers appendix.

Similarly, this book concludes with a Final Exam in the first appendix, and the answers to that also found in the second appendix.

How to Read this Book

I have organized this book to be read from beginning to end. While this may seem patently obvious, my students often express legitimate frustration about books (or teachers) that, in discussing a programming concept, mention other concepts that are covered several chapters later or, even worse, not at all. Therefore, I have endeavored to present the material in a linear, logical progression. This not only avoids the frustration of material that is out of order, but also enables you in each succeeding chapter to build on the skills you learned in the preceding chapters.

Special Features

Throughout each chapter are Notes, Tips, and Cautions, as well as detailed code listings. To provide you with additional opportunities to review, there is a Quiz at the end of each chapter and a Final Exam (found in the first appendix) at the end of this book. Answers to both are contained in the following appendix.

The overall objective is to get you up to speed quickly, without a lot of dry theory or unnecessary detail. So let's get started. It's easy and fun to write C++ programs.

Contacting the Author

Hmmm... it depends why. Just kidding. While I always welcome gushing praise and shameless flattery, comments, suggestions, and yes, even criticism also can be valuable. The best way to contact me is via e-mail; you can use jkent@genghiskhent.com (the domain name is based on my students' fond nickname for me). Alternately, you can visit my web site, http://www.genghiskhent.com/. Don't be thrown off by the entry page; I use this site primarily to support the online classes and online components of other classes that I teach at the college, but there will be a link to the section that supports this book.

I hope you enjoy this book as much as I enjoyed writing it.

CHAPTER 1

How a C++ Program Works

You probably interact with computer programs many times during an average day. When you arrive at work and find out your computer doesn't work, you call tech support. At the other end of the telephone line, a computer program forces you to navigate a voicemail menu maze and then tortures you while you are on perpetual hold with repeated insincere messages about how important your call is, along with false promises about how soon you will get through.

When you're finally done with tech support, you decide to take a break and log on to your now-working computer to do battle with giant alien insects from the planet Megazoid. Unfortunately, the network administrator catches you goofing off using yet another computer program which monitors employee computer usage. Assuming you are still employed, an accounts payable program then generates your payroll check.

On your way home, you decide you need some cash and stop at an ATM, where a computer program confirms (hopefully) you have enough money in your bank account and then instructs the machine to dispense the requested cash and (unfortunately) deducts that same amount from your account.

Most people, when they interact with computers as part of their daily routine, don't need to consider what a computer program is or how it works. However, a computer programmer should know the answers to these and related questions, such as what is a programming language, and how does a C++ program actually work? When you have completed this chapter, you will know the answers to these questions, and also understand how to create and run your own computer program.

What Is a Computer Program?

Computers are so widespread in our society because they have three advantages over us humans. First, computers can store huge amounts of information. Second, they can recall that information quickly and accurately. Third, computers can perform calculations with lightning speed and perfect accuracy.

The advantages that computers have over us even extend to thinking sports like chess. In 1997, the computer Deep Blue beat the world chess champion, Garry Kasparov, in a chess match. In 2003, Kasparov was out for revenge against another computer, Deep Junior, but only drew the match. Kasparov, while perhaps the best chess player ever, is only human, and therefore no match for the computer's ability to calculate and remember prior games.

However, we have one very significant advantage over computers. We think on our own, while computers don't, at least not yet anyway. Indeed, computers fundamentally are far more brawn than brain. A computer cannot do anything without step-by-step instructions from us telling it what to do. These instructions are called a computer program, and of course are written by a human, namely a computer programmer. Computer programs enable us to harness the computer's tremendous power.

What Is a Programming Language?

When you enter a darkened room and want to see what is inside, you turn on a light switch. When you leave the room, you turn the light switch off.

The first computers were not too different than that light switch. These early computers consisted of wires and switches in which the electrical current followed a path dependent on which switches were in the on (one) or off (zero) position. Indeed, I built such a simple computer when I was a kid (which according to my own children was back when dinosaurs still ruled the earth).

Each switch's position could be expressed as a number: 1 for the on position, 0 for the off position. Thus, the instructions given to these first computers, in the form of the switches' positions, essentially were a series of ones and zeroes.

Today's computers, of course, are far more powerful and sophisticated than these early computers. However, the language that computers understand, called machine language, remains the same, essentially ones and zeroes.

While computers think in ones and zeroes, the humans who write computer programs usually don't. Additionally, a complex program may consist of thousands or even millions of step-by-step machine language instructions, which would require an inordinately long amount of time to write. This is an important consideration since, due to competitive market forces, the amount of time within which a program has to be written is becoming increasingly less and less.

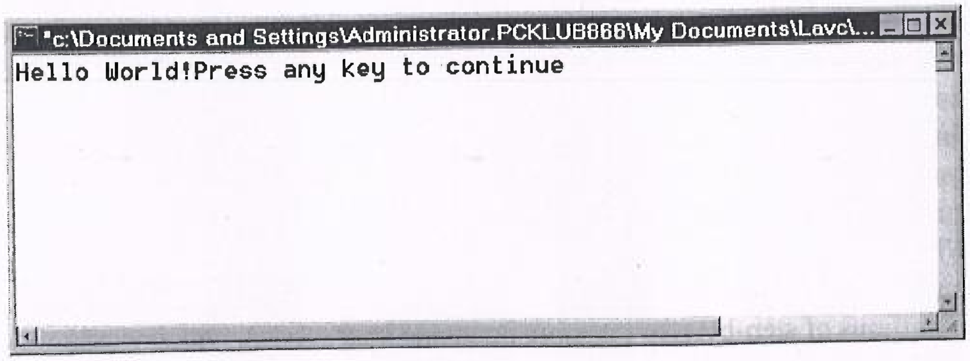
Fortunately, we do not have to write instructions to computers in machine language. Instead, we can write instructions in a programming language. Programming languages are far more understandable to programmers than machine language because programming languages resemble the structure and syntax of human language, not ones and zeroes. Additionally, code can be written much faster with programming languages than machine language because programming languages automate instructions; one programming language instruction can cover many machine language instructions.

C++ is but one of many programming languages. Other popular programming languages include Java, C#, and Visual Basic. There are many others. Indeed, new languages are being created all the time. However, all programming languages have essentially the same purpose, which is to enable a human programmer to give instructions to a computer.

Why learn C++ instead of another programming language? First, it is very widely used, both in industry and in education. Second, many other programming languages, including Java and C#, are based on C++. Indeed, the Java programming language was written using C++. Therefore, knowing C++ makes learning other programming languages easier.

Anatomy of a C++ Program

It seems to be a tradition in C++ programming books for the first code example to output to a console window the message "Hello World!" (shown in Figure 1-1).



"c:\Documents and Settings\Administrator.PCKLUB866\My Documents\Lavc\...

```
Hello World!Press any key to continue
```

Figure 1-1 C++ program outputting "Hello World!" to the screen

---

NOTE: *The term "console" goes back to the days before Windows when the screen did not have menus and toolbars but just text. If you have typed commands using DOS or UNIX, you likely did so in a console window. The text "Press any key to continue" immediately following "Hello World!" is not part of the program, but instead is a cue for how to close the console window.*

Unfortunately, all too often the "Hello World!" example is followed quickly by many other program examples without the book or teacher first stopping to explain how the "Hello World!" program works. The result soon is a confused reader or student who's ready to say "Goodbye, Cruel World."

While the "Hello World!" program looks simple, there actually is a lot going on behind the scenes of this program. Accordingly, we are going to go through the following code for the "Hello World!" program line by line, though not in top-to-bottom order.

```
#include <iostream>
using namespace std;

int main(void)
{
   cout << "Hello World!";
   return 0;
}
```

NOTE: *The code a programmer writes is referred to as source code, which is saved in a file that usually has a .cpp extension, standing for C++.*

The main Function

As discussed in the "What Is a Programming Language?" section, the purpose of C++, or any programming language, is to enable a programmer to write instructions for a computer. Often, a task is too complex for just one instruction. Instead, several related instructions are required.

A *function* is a group of related instructions, also called statements, which together perform a particular task. The name of the function is how you refer to these related statements. In the "Hello World!" program, *main* is the name of a function. A program may have many functions, and in Chapter 9 I will show you how to create and use functions. However, a program must have one main function, and only one main function. The reason is that the main function is the starting point for every C++ program. If there was no main function, the computer would not know where to start the program. If there was more than one main function, the program would not know whether to start at one or the other.

Note: The main function is preceded by int and followed by void in parentheses. We will cover the meaning of both in Chapter 9.

The Function Body

Each of the related instructions, or statements, which belong to the main function are contained within the *body* of that function. A function body starts with a left curly brace, {, and ends with a right curly brace, }.

Each statement usually ends with a semicolon. The main function has two statements:

```
cout << "Hello World!";
return 0;
```

Statements are executed in order, from top to bottom. Don't worry, the term "executed" doesn't mean the statement is put to death. Rather, it means that the statement is carried out, or executed, by the computer.

cout

The first statement is

```
cout << "Hello World!";
```

cout is pronounced "C-out." The "out" refers to the direction in which cout sends a stream of data.

A data stream may flow in one of two directions. One direction is input—into your program from an outside source such as a file or user keyboard input. The other direction is output—out from your program to an outside source such as a monitor, printer, or file.

cout concerns the output stream. It sends information to the standard output device. The standard output device usually is your monitor, though it can be something else, such as a printer or a file on your hard drive.

The << following cout is an operator. You likely have used operators before, such as the arithmetic operators +, –, *, and /, for addition, subtraction, multiplication, and division, respectively.

The << operator is known as the stream insertion operator. It inserts the information immediately to its right—in this example, the text "Hello World!" into the data stream. The cout object then sends that information to the standard output device—in this case, the monitor.

NOTE: In Chapter 3, you will learn about the counterparts to the cout object and the << operator, the cin object, which concerns the input stream, and the >> operator used with the cin object.

The return 0 Statement

The second and final statement returns a value of zero to the computer's operating system, whether Windows, UNIX, or another. This tells the operating system that the program ended normally. Sometimes programs do not end normally, but instead crash, such as if you run out of memory during the running of the program. The operating system may need to handle this abnormal program termination differently than normal termination. That is why the program tells the operating system that this time it ended normally.

The #include Directive

Your C++ program "knows" to start at the main function because the main function is part of the core of the C++ language. We certainly did not write any code that told the C++ program to start at *main*.

Similarly, your C++ program seems to know that the cout object, in conjunction with the stream insertion operator <<, outputs information to the monitor. We did not write any code to have the cout object and the << operator achieve this result.

However, the cout object is not part of the C++ core language. Rather, it is defined elsewhere, in a *standard library file*. C++ has a number of standard library files, each defining commonly used objects. Outputting information to the monitor certainly is a common task. While you could go to the trouble of writing your own function that outputs information to the screen, a standard library file's implementation of cout saves you the trouble of "reinventing the wheel."

While C++ already has implemented the cout object for you in a standard library file, you still have to tell the program to include that standard library file in your application. You do so with the #include directive, followed by the name of the library file. If the library file is a standard library file, as opposed to one you wrote (yes, you can create your own), then the file name is enclosed in angle brackets, < and >.

The cout object is defined in the standard library file *iostream*. The "io" in iostream refers to input and output—"stream" to a stream of data. To use the cout object, we need to include the iostream standard library file in our application. We do so with the following include directive:

```
#include <iostream>
```

The include directive is called a *preprocessor directive*. The preprocessor, together with the compiler and linker, are discussed later in this chapter in the section "Translating the Code for the Computer." The preprocessor directive, unlike statements, is not ended by a semicolon.

Namespace

The final statement to be discussed in the Hello World! example is

```
using namespace std;
```

C++ uses *namespaces* to organize different names used in programs. Every name used in the iostream standard library file is part of a namespace called *std*. Consequently, the cout object is really called std::cout. The using namespace std statement avoids the need for putting std:: before every reference to cout, so we can just use cout in our code.

Translating the Code for the Computer

While you now understand the "Hello World!" code, the computer won't. Computers don't understand C++ or any other programming language. They understand only machine language.

Three programs are used to translate your source code into an *executable* file that the computer can run. These programs are, in their order of appearance:

1. Preprocessor
2. Compiler
3. Linker

Preprocessor

The preprocessor is a program that scans the source code for preprocessor directives such as include directives. The preprocessor inserts into the source code all files included by the include directives.

In this example, the iostream standard library file is included by an include directive. Therefore, the preprocessor directive inserts the contents of that standard library file, including its definition of the cout object, into the source code file.

Compiler

The compiler is another program that translates the preprocessed source code (the source code after the insertions made by the preprocessor) into corresponding machine language instructions, which are stored in a separate file, called an object file, having an .obj extension. There are different compilers for different programming languages, but the purpose of the compiler is essentially the same, the translation of a programming language into machine language, no matter which programming language is involved.

The compiler can understand your code and translate it into machine language only if your code is in the proper syntax for that programming language. C++, like other programming languages, and indeed most human languages, has rules for the spelling of words and for the grammar of statements. If there is a syntax error, then the compiler cannot translate your code into machine language instructions, and instead will call your attention to the syntax errors. Thus, in a sense, the compiler acts as a spell checker and grammar checker.

Linker

While the object file has machine language instructions, the computer cannot run the object file as a program. The reason is that C++ also needs to use another code library, called the run-time library, for common operations, such as the translation of keyboard input or the ability to interact with external hardware such as the monitor to display a message.

NOTE: The run-time library files may already be installed as part of your operating system. If not, you can download the run-time library files from Microsoft or another vendor. Finally, if you install an IDE as discussed in the next section, the run-time library files are included with the installation.

The linker is a third program that combines the object file with the necessary parts of the run-time library. The result is the creation of an executable file with an .exe extension. The computer runs this file to display "Hello World!" on the screen.

Using an IDE to Create and Run the "Hello World!" Project

You can use any plain-text editor such as Notepad to write the source code. You also can download a free compiler, which usually includes a preprocessor and linker. You then can compile and run your code from the *command line.* The command line may be, for example, a DOS prompt at which you type a command that specifies the action you want, such as compiling, followed by the name of the file you want to compile.

While there is nothing wrong with using a plain-text editor and command line tools, many programmers, including me, prefer to create, compile, and run their programs in a C++ Integrated Development Environment, known by the acronym IDE. The term "integrated" in IDE means that the text editor, preprocessor, compiler, and linker are all together under one (software) roof. Thus, the IDE enables you to create, compile, and run your code using one program rather than separate programs. Additionally, most IDEs have a graphical user interface (GUI) that makes them easier for many to use than a command line. Finally, many IDEs have added features that ease your task of finding and fixing errors in your code.

The primary disadvantage of using IDEs is you have to pay to purchase them (though there are some free ones). They also require additional hard drive space and memory. Nevertheless, I recommend obtaining an IDE since it enables you to focus on C++ programming issues without distractions such as figuring out the right commands to use on the command line.

There are several good IDEs on the market. Microsoft's, called Visual C++, can be obtained separately or as part of Microsoft's Visual Studio product. Borland offers

C++ Builder, both in a free and commercial version. IBM has a VisualAge C++ IDE. There are a number of others as well.

In this book, I will use Microsoft's Visual C++ .NET 2003 IDE since I happen to have it. However, most IDEs work essentially the same way, and your code will compile and run the same no matter which IDE you use as long as you don't use any library files custom to a particular IDE. The standard library files we will be using, such as iostream, are the same in all C++ IDEs.

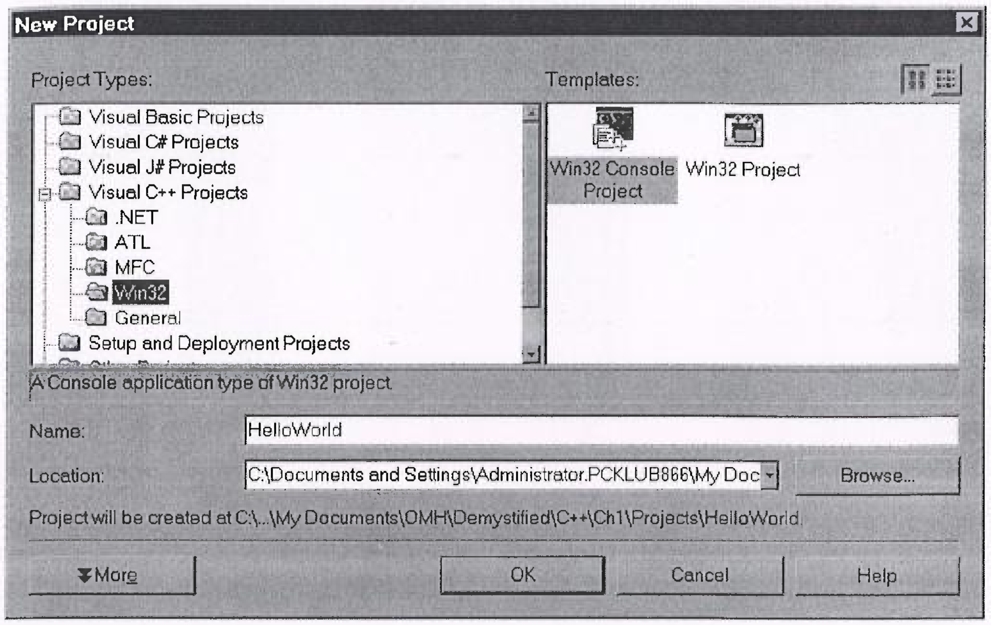
Additionally, I am running the code on a Windows 2000 operating system. The results should be similar on other operating systems, not just Windows operating systems, but additional types of operating systems as well, such as UNIX.

Let's now use the IDE to write the source code for the "Hello World!" project, and then compile and run it.

Setting Up the "Hello World!" Project

Once you have purchased and installed Visual C++ .NET 2003, either as a standalone application or as part of Visual Studio .NET 2003, you are now ready to start your first project, which is to create and run the "Hello World!" application.

1. Start Visual C++.
2. Open the New Project dialog box shown in Figure 1-2 using the File | New | Project menu command. (The values in the Name and Location fields will be set in steps 5 and 6.)
3. In the left or list pane of the New Project dialog box, choose Visual C++ Projects from the list of Project Types, and then the Win32 subfolder, as shown in Figure 1-2.
4. In the right or contents pane of the New Project dialog box, choose Win32 Console Project from the list of templates. The word console comes from the application running from a console window. Win32 comes from the Windows 32-bit operating system, such as Windows 9*x*, 2000, or XP.
5. In the Location field, using the Browse button, choose an existing folder under which you will create the subfolder where you will put your project.
6. In the Name field, type the name you've chosen for your project. This will also be the name of the subfolder created to store your project files. I suggest you use a name that describes your project so you can locate it more easily later.
7. Click the OK button. This will display the Win32 Application Wizard, shown in Figure 1-3.



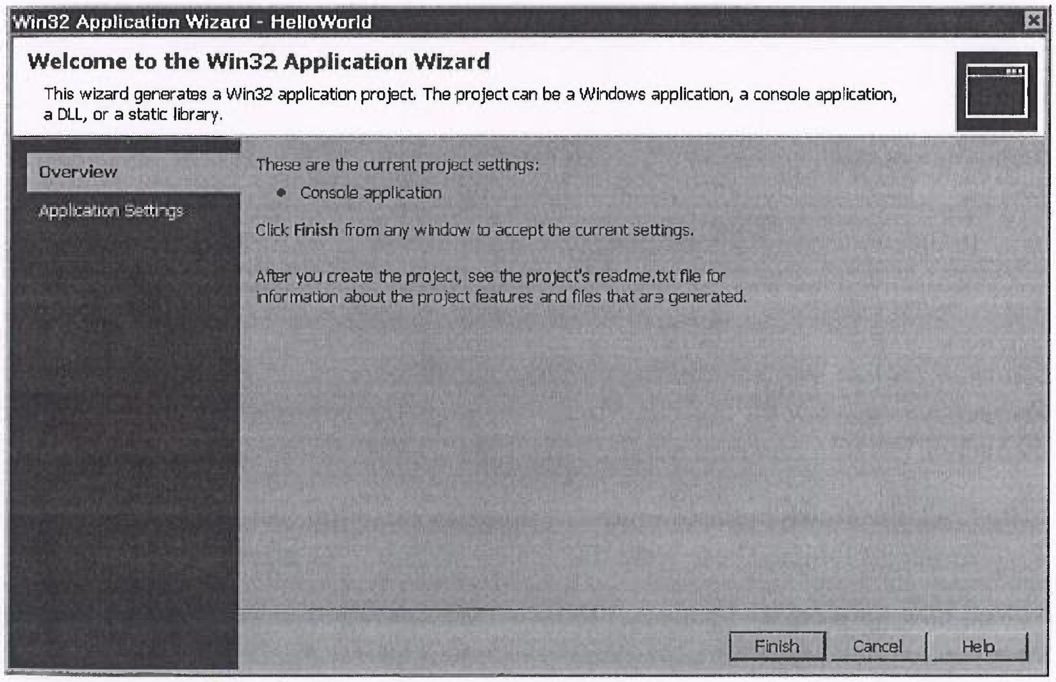
A Console application type of Win32 project

Name: HelloWorld

Location: C:\Documents and Settings\Administrator.PCKLUB866\My Doc

Project will be created at C:\...\My Documents\OMH\Demystified\C++\Ch1\Projects\HelloWorld.

Figure 1-2 Creating a New Project



Welcome to the Win32 Application Wizard

This wizard generates a Win32 application project. The project can be a Windows application, a console application, a DLL, or a static library.

These are the current project settings:

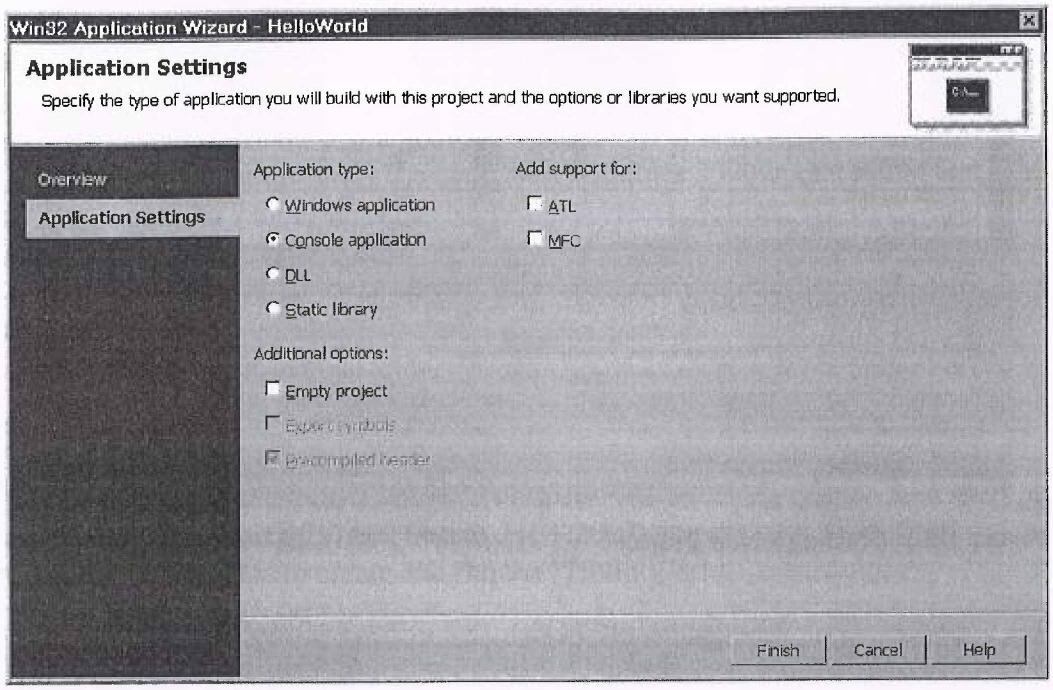
- Console application

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

Figure 1-3 Starting the Win32 Application Wizard

8. Click the Application Settings menu item on the left. The appearance of the Win32 Application Wizard then changes to that shown in Figure 1-4.


Win32 Application Wizard - HelloWorld

Application Settings

Specify the type of application you will build with this project and the options or libraries you want supported.

Application type:
- ( ) Windows application
- (•) Console application
- ( ) DLL
- ( ) Static library

Add support for:
- [ ] ATL
- [ ] MFC

Additional options:
- [ ] Empty project
- [ ] Export symbols
- [✓] Precompiled header

Figure 1-4 Win32 Application Wizard after choosing Application Settings

9. Choose, if necessary, Console Application under Application Type (this is the default) and Empty Project under Additional Options. Choosing Empty Project will disable both checkboxes under Add Support For, which should be disabled anyway.

CAUTION: *Make sure you follow this step carefully, particularly choosing Empty Project, which is not the default. Not configuring Application Settings properly is a common mistake and may require you to start over.*

10. Click the Finish button. Figure 1-5 shows the new subfolder HelloWorld and its parent folder. These were the name and location chosen in steps 5 and 6.

You now have created a project for your application. The project is a shell for your application, containing files that will support the creation and running of your application. However, right now the project is empty of any code you have written, so it won't do anything. Accordingly, the next step is to start writing code.

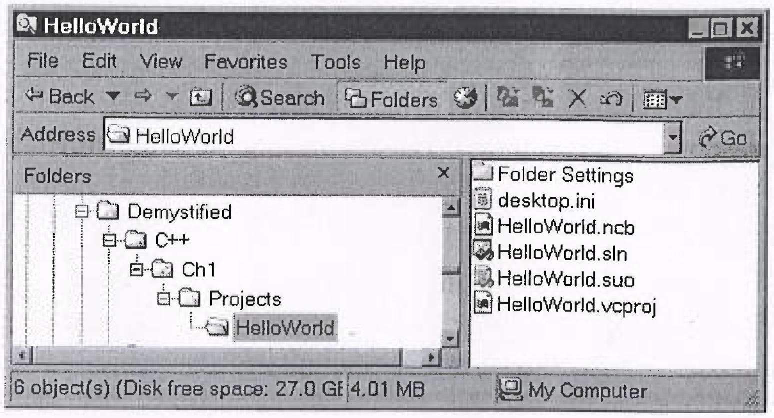


Figure 1-5 Windows Explorer showing newly created subfolder and files

Writing the Source Code

Visual C++ has a view of a project that is similar to Windows Explorer. That view is called Solution Explorer, shown in Figure 1-6. If Solution Explorer is not already displayed, you can display it with the menu command View | Solution Explorer.

Solution Explorer has folders for both source and header files. The file in which the code for the "Hello World!" application will be written is a source file. Source

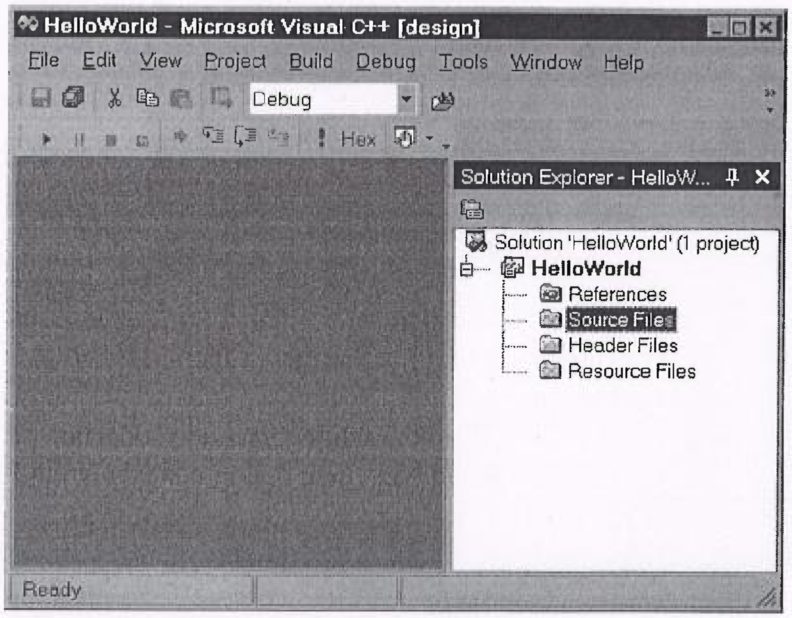

HelloWorld - Microsoft Visual C++ [design]
File Edit View Project Build Debug Tools Window Help
Debug
Hex
Solution Explorer - HelloW...
Solution 'HelloWorld' (1 project)
HelloWorld
References
Source Files
Header Files
Resource Files
Ready

Figure 1-6 Viewing your project with Solution Explorer

files have a .cpp extension, cpp standing for C++. By contrast, the iostream file that is included by the include directive is a header file. Header files have an .h extension—the h standing for header.

We will use Solution Explorer to add a new source file to the project, after which we will write code in that new source file.

You can use the following steps to add a new source file to the project:

1. Right-click Source Files in Solution Explorer. This will display a shortcut menu, shown in Figure 1-7.

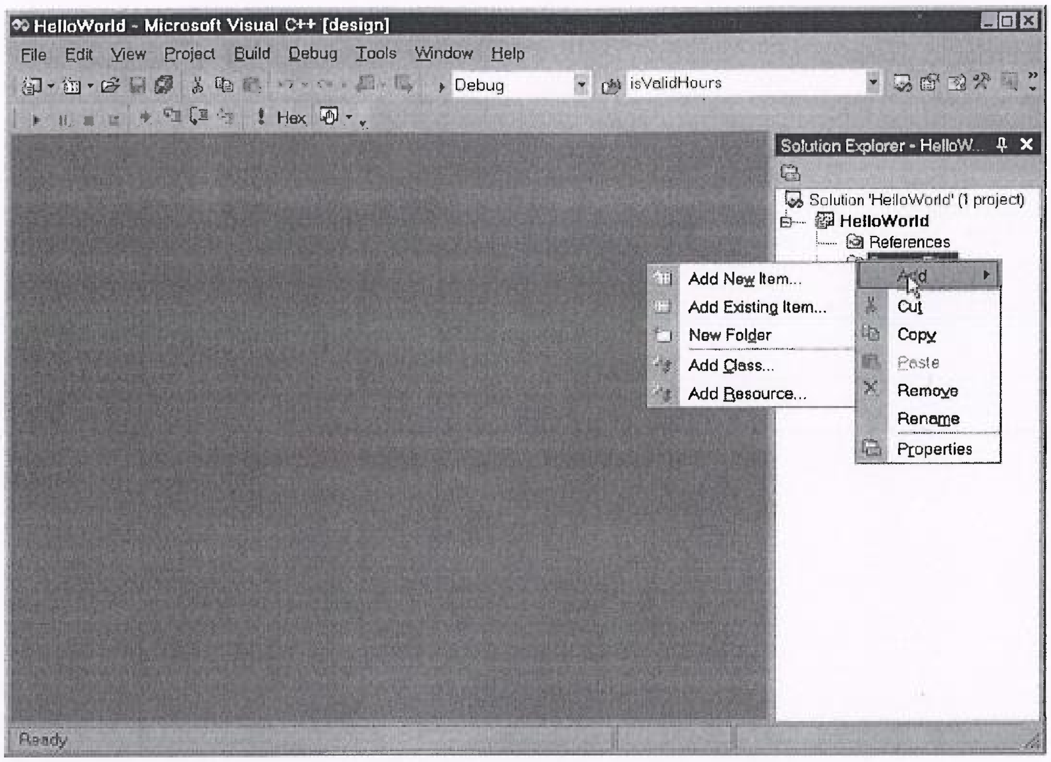

HelloWorld - Microsoft Visual C++ [design]
File Edit View Project Build Debug Tools Window Help
Debug
isValidHours
Hex
Solution Explorer - HelloW...
Solution 'HelloWorld' (1 project)
HelloWorld
References
Add New Item...
Add Existing Item...
New Folder
Add Class...
Add Resource...
Add
Cut
Copy
Paste
Remove
Rename
Properties
Ready

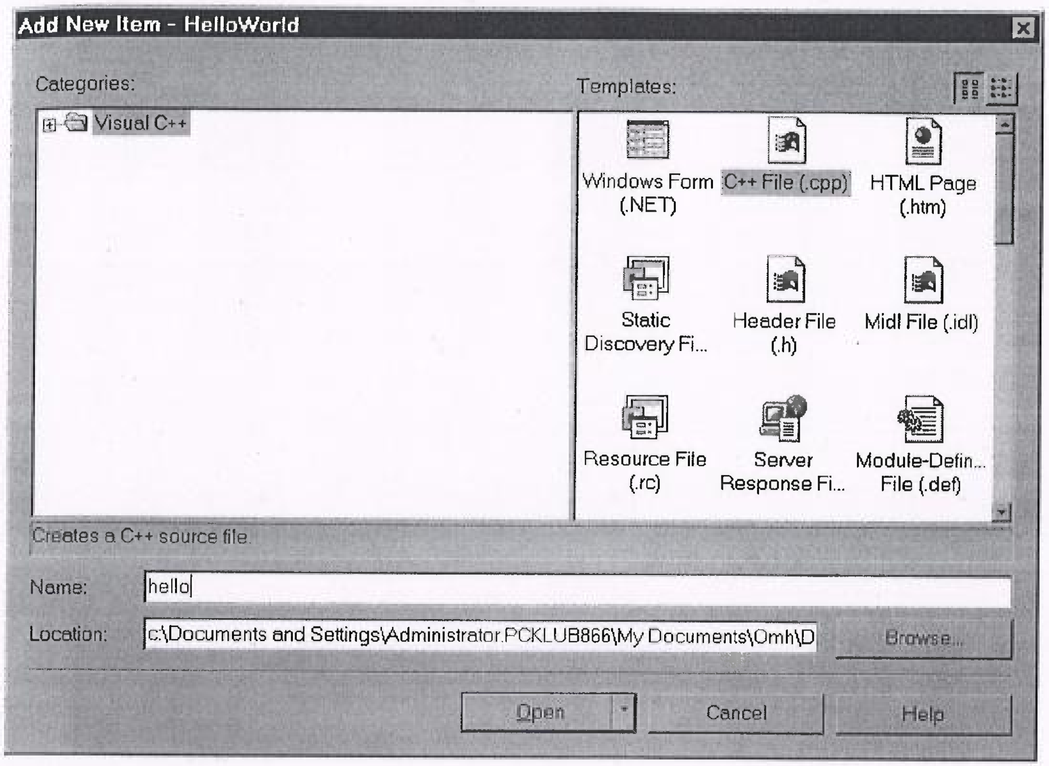
Figure 1-7 Source Files shortcut menu

2. Choose Add | Add New Item from the shortcut menu to add a new source to the project. This will display the Add New Item dialog box, shown in Figure 1-8.

NOTE: *If the source file already exists, you can add it to your project using the Add | Add Existing Item shortcut menu item.*

3. Generally, you will not change the Location field, which is the subfolder in which the project files are stored. Type the name of the new source file in the

Name field. You do not need to type the .cpp extension; that extension will be appended automatically since it is a source file. By typing **hello**, as shown in Figure 1-8, the new file will be called hello.cpp.



Creates a C++ source file.

Name: hello

Location: c:\Documents and Settings\Administrator.PCKLUB866\My Documents\Omh\D

Figure 1-8 Adding a New Source File to your Project

4. When you are done, click the Open button. Figure 1-9 shows the new hello.cpp file in Solution Explorer.

Writing the code is easy. Double-click hello.cpp in Solution Explorer. As shown in Figure 1-10, this will display the hello.cpp file, which at this point is blank.

Now just type your code. When finished, hello.cpp should appear as in Figure 1-11.

Caution: You also can use Notepad or any other text editor to write the code. However, do not use Microsoft Word or any other word processing program to write your code. While a word processing program enables you to neatly format your code, it does so using hidden formatting characters that the compiler does not understand and will regard as syntax errors.

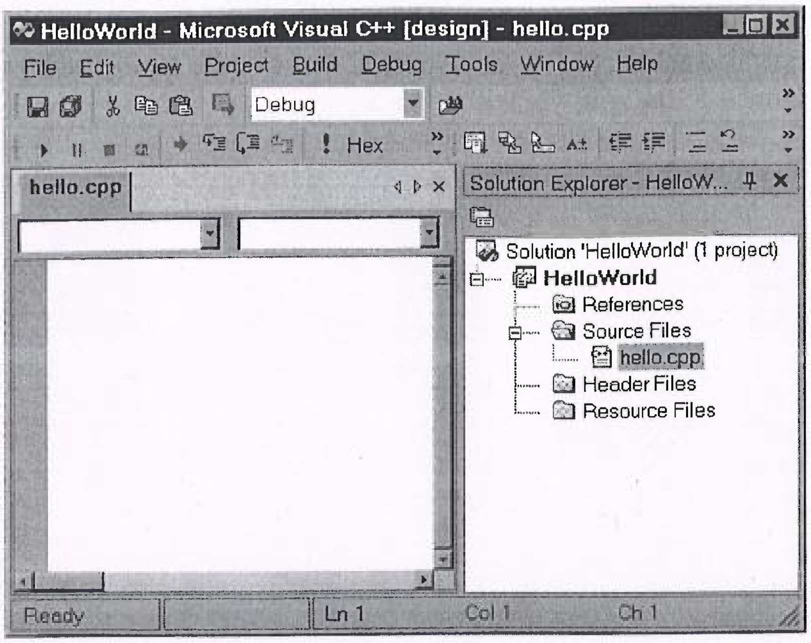

HelloWorld - Microsoft Visual C++ [design] - hello.cpp
File Edit View Project Build Debug Tools Window Help
Debug
Hex
hello.cpp
Solution Explorer - HelloW...
Solution 'HelloWorld' (1 project)
HelloWorld
References
Source Files
hello.cpp
Header Files
Resource Files
Ready
Ln 1
Col 1
Ch 1

Figure 1-9 Solution Explorer showing the new .cpp file

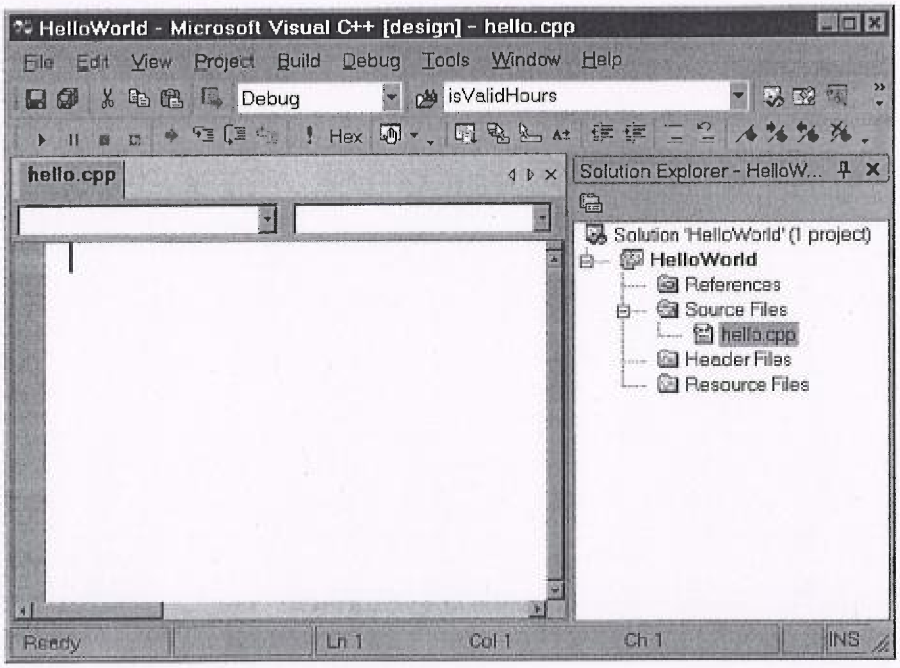
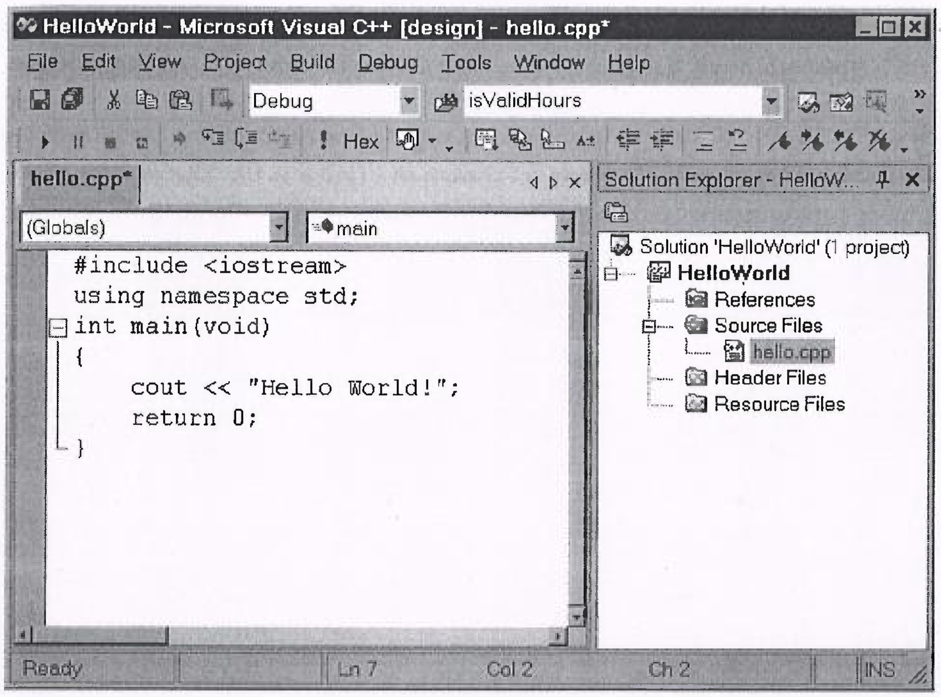

HelloWorld - Microsoft Visual C++ [design] - hello.cpp
File Edit View Project Build Debug Tools Window Help
Debug
isValidHours
Hex
hello.cpp
Solution Explorer - HelloW...
Solution 'HelloWorld' (1 project)
HelloWorld
References
Source Files
hello.cpp
Header Files
Resource Files
Ready
Ln 1
Col 1
Ch 1
INS

Figure 1-10 The source file before typing code


HelloWorld - Microsoft Visual C++ [design] - hello.cpp*
File Edit View Project Build Debug Tools Window Help
Debug
isValidHours
Hex
hello.cpp*
(Globals)
main
Solution Explorer - HelloW...
Solution 'HelloWorld' (1 project)
HelloWorld
References
Source Files
hello.cpp
Header Files
Resource Files
Ready
Ln 7
Col 2
Ch 2
INS

```
#include <iostream>
using namespace std;
int main(void)
{
    cout << "Hello World!";
    return 0;
}
```

Figure 1-11 The source file after typing code

Save your work, such as by pressing the Save toolbar button. We're now ready to compile.

Building the Project

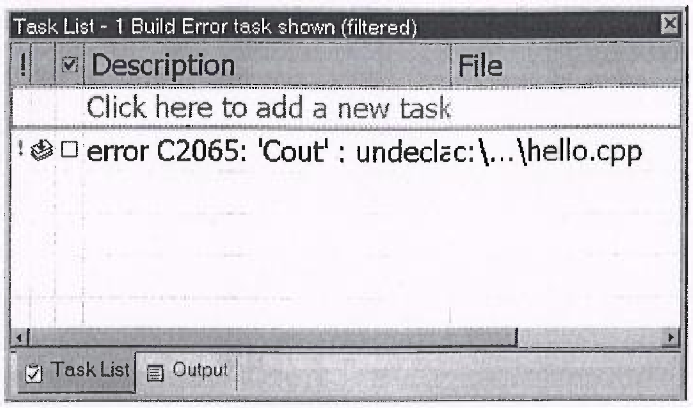
You compile your code from the Build menu. You may compile your code from any one of the following different menu choices:

- Build | Solution
- Rebuild | Solution
- Build | HelloWorld
- Rebuild | HelloWorld

HelloWorld is the name of your project. A solution may contain more than one project. Here the solution contains only one project, so there is no practical difference between the project and the solution.

Build means to compile changes from the last compilation (if there was one). Rebuild means to start compilation from the beginning. Build therefore is usually faster, but Rebuild is used when there have been extensive changes since the last compilation. As a practical matter, it rarely makes a difference which one you choose.

Before we compile, make one change to the code, changing cout to Cout (capitalizing the C). Then choose one of the four compilation options. A Task List window should display, noting a build error, as shown in Figure 1-12. The error description in the Task List window is "error C2065: 'Cout' : undeclared identifier."


Task List - 1 Build Error task shown (filtered)

!	☑	Description	File
		Click here to add a new task	
!	☐	error C2065: 'Cout' : undecl	c:\...\hello.cpp

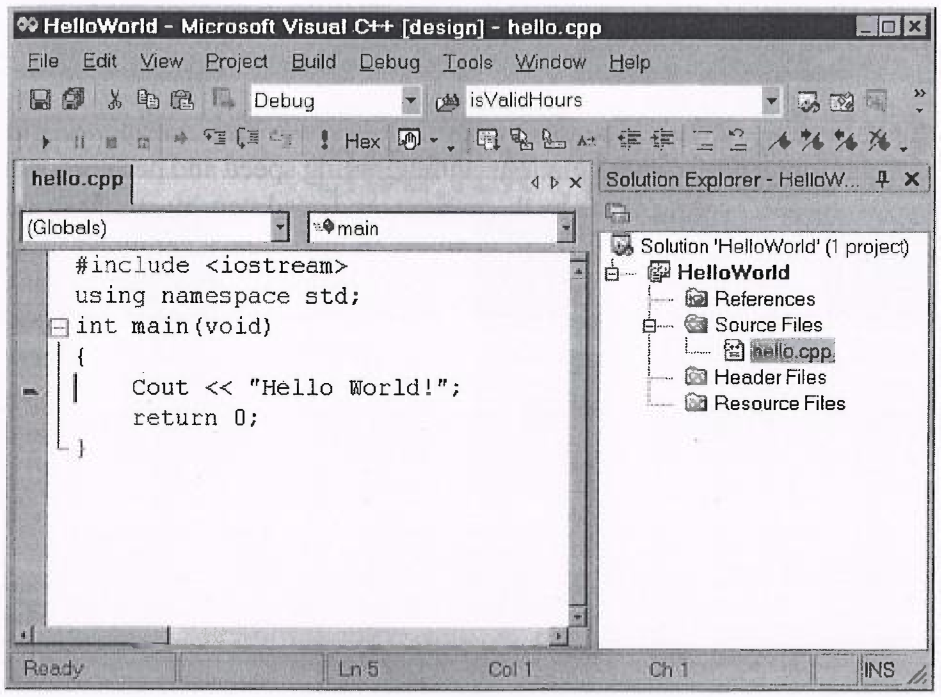
Figure 1-12 The Task List window showing a compilation error

TIP: If the description column is not wide enough to show the entire error description, you can display the error description in a pop-up window by right-clicking the error description and choosing Show Description Tooltip from the shortcut menu.

As explained in the earlier section on the Compiler, the compiler can understand your code and translate it into machine language only if your code is in the proper syntax for that programming language. As also explained there, C++ has rules for the spelling of words and for the grammar of statements. If there is a violation of those rules, that is, a syntax error, then the compiler cannot translate your code into machine language instructions, and instead will call your attention to the syntax errors.

In C++, code is case sensitive. That is, a word capitalized is not the same as the word uncapitalized. The correct spelling is cout; Cout is wrong. Since C++ does not know what Cout is, you get the error message that it is an "undeclared identifier."

While here the code is short, if your code is quite lengthy, it is not easy to spot where the error is in the code. If you double-click the error in the Task List window, then a cursor will blink at the line where Cout is, and an icon will display in the margin (as shown in Figure 1-13).


HelloWorld - Microsoft Visual C++ [design] - hello.cpp
File Edit View Project Build Debug Tools Window Help
Debug
isValidHours
Hex
hello.cpp
(Globals)
main
Solution Explorer - HelloW...
Solution 'HelloWorld' (1 project)
HelloWorld
References
Source Files
hello.cpp
Header Files
Resource Files
Ready
Ln 5
Col 1
Ch 1
INS

```
#include <iostream>
using namespace std;
int main(void)
{
    Cout << "Hello World!";
    return 0;
}
```

Figure 1-13 The error highlighted in the code window

Now change Cout to cout, and then compile your code again. This time compilation should be successful. Using Windows Explorer, you can now see in the Debug subfolder of your HelloWorld project folder a file called hello.obj and another file called hello.exe. These are the object and executable files previously discussed in the section "Translating the Code for the Computer." Accordingly, building the project involved the preprocessor, the compiler, and the linker.

Running the Code

The final step is to run the code. You do so from the Debug menu. You may choose either Debug | Start or Debug | Start Without Debugging. The difference is whether you wish to use the debugger, an issue which we will discuss in a later chapter. Since we are not going to use the debugger this time, choose Debug | Start Without Debugging as it is slightly faster. The result is the console window displaying "Hello World!" (shown way back in Figure 1-1).

CHAPTER 2

Memory and Data Types

After I wrote my first book, I expectantly waited every day for my mail, hoping to receive requests for my autograph. The result was proof of the adage "be careful what you ask for." My mailbox was stuffed with numerous requests for my autograph. Alas, these requests came from those who wanted to share my money, not my fame. My autograph was requested on checks to pay my mortgage, credit cards, insurance, phone service, electricity; well, you get the picture.

These companies who love sending me bills could not possibly keep track of their thousands of customers by using pencil and paper. Instead, they use computer programs, which harness the computer's ability to store very large amounts of information and to retrieve that stored information very quickly.

We use our memory to store and recall information. So do computers. However, a computer's memory is very different from ours. This chapter will explain how a computer's memory works.

Information, also called data, comes in different forms. Some data is numeric, such as the amount of my gas bill. Other data is text, such as my name on my gas bill. The type of data, whether numeric, text, or something else, quite logically is referred to as the "data type." The data type you choose will affect not only the form in which the data is stored, but also the amount of memory required to store it. This chapter will explain the different data types.

Memory

Computer programs consist of instructions and data. As discussed in Chapter 1, instructions, written in a programming language such as C++ and then translated by the compiler and linker into machine language, give the computer step-by-step directions on what to do. The data is the information that is the subject of the program. For example, if the user of your computer program wants a list of all students with a GPA of 4.0, the data could be a list of all students and their GPAs. The program then would follow instructions to determine and output the list of all students with a GPA of 4.0.

The computer program's instructions and data have to be in the computer's memory for the program to work. This section will explain the different types of computer memory, as well as how and where instructions and data are stored in computer memory.

Types of Memory

There are three principal memory locations on your computer.

- The central processing unit (CPU)
- Random access memory (RAM)
- Persistent storage

Cache Memory

The CPU is the brains of the computer. You may have thought about the CPU when you last considered purchasing a computer, since the CPU's speed often is an important purchase consideration. The faster the CPU's speed, the faster your computer runs.

NOTE: A hertz, named after Heinrich Hertz, who first detected electromagnetic waves, represents one cycle per second. CPU speed is measured in megahertz (MHz), which represents one million cycles per second, or gigahertz (GHz), which represents 1 billion cycles per second. For example, a CPU that runs at 800 MHz executes 800 million cycles per second. Each computer instruction requires a fixed number of cycles, so the CPU speed determines how many instructions per second the CPU can execute.

The CPU, in addition to coordinating the computer's operations, also has memory, called *cache memory*. The CPU's cache memory includes a segment called a *register*. This memory is used to store frequently used instructions and data.

The CPU can access cache memory extremely quickly because it doesn't have far to go; the memory is right on the CPU. However, the amount of available cache memory is quite small; there is only enough room for the most frequently used instructions and data. The remainder of the instructions and data have to be stored somewhere else.

Random Access Memory

That somewhere else is *random access memory,* or RAM. You may also have considered RAM when you last purchased a computer, since the more RAM a computer has, the more programs it can run at one time, and the faster it runs.

The CPU can access RAM almost as quickly as cache memory. Additionally, the amount of RAM available to store instructions and data is much larger than the amount of available cache memory.

However, RAM, like cache memory, is temporary. Instructions and data contained in main memory are lost once the computer is powered down. You may have had the unpleasant experience of losing unsaved data when your computer powered off during a power failure, or had to be rebooted.

Additionally, we would want the data to remain intact after the program ended, even if the computer is rebooted or powered off. That is not possible with RAM.

Furthermore, your computer likely has many other programs, for e-mail, Internet, word processing, and so on, that you may not be using right now, but you may want to use in the future. Likewise, your computer also may have other data files, such as term papers, letters, tax spreadsheets, e-mail messages, and so on, that you also may not be using right now, but that you may want to use in the future. Accordingly, we need another memory location, which unlike cache memory or RAM, is persistent—that is, it will persist even though the computer is rebooted or turned off.

Persistent Storage

That other, persistent type of computer memory is called, naturally enough, *persistent storage*. This usually is a hard drive, but also could be, among other devices, a CD-ROM or DVD-ROM, floppy or zip disk, or optical drive. However, no matter what storage device is used, persistent storage is lasting; instructions and data remain stored even when the computer is powered down. Thus, your computer can be turned off for months, but when it is turned on, the files you previously saved are still there.

Persistent storage, in addition to being lasting, also has a much larger capacity than RAM—about one hundred to one thousand times larger.

Since persistent storage is lasting and has a very large capacity, it is used to store both programs and data. For example, if you installed Microsoft Word on your computer, the files for this program would be stored on your hard drive. If you then prepared documents using that program, those documents likewise would be saved as files on your hard drive.

While persistent storage has the advantages of being lasting and having a large capacity, a computer program cannot execute instructions located in persistent storage. The instructions must be loaded from persistent storage into RAM. Similarly, a computer program cannot manipulate data located in persistent storage. This data likewise must be loaded from persistent storage into RAM.

NOTE: *While beyond the scope of this chapter, persistent storage also can serve as a backup to RAM, and when serving this purpose is called virtual memory or swap space.*

Generally, computer programs use RAM to store instructions and data, so RAM will be our focus in discussing memory. However, much of the discussion of memory also may apply to persistent storage. CPU cache memory is a different subject, discussed more in connection with programming languages, such as assembly language, that are far closer to machine language than is C++.

Addresses

When someone asks where you live, you may answer 1313 Mockingbird Lane. That is your address.

Addresses are used to locate persons or places. Addresses usually follow a logical pattern. For example, the addresses on one block may be from 1300 to 1399, the next from 1400 to 1499, and so on.

Locations in memory also are identified by address. These addresses often look quite different than the street addresses we're used to, since they usually are expressed as hexadecimal (Base 16) numbers such as 0x8fc1. However, regardless of how the number is written, as shown in Figure 2-1, memory addresses follow the same logical, sequential pattern as do street addresses, one number coming after another.

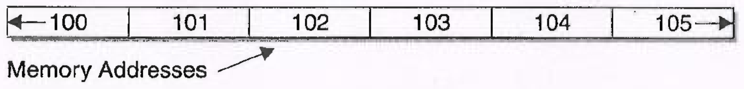


Figure 2-1 Sequence of memory addresses

NOTE: ***Hexadecimal Numbers****—We usually use numbers that are decimal, or Base 10, in which each digit is between 0 and 9. By contrast, memory addresses usually are expressed as hexadecimal, or Base 16, in which each digit is between 1 and 15. Since 10, 11, 12, 13, 14, and 15 are not single digits, 10 is expressed as a, 11 as b, 12 as c, 13 as d, 14 as e, and 15 as f. The number 16 in decimal is expressed as 10 in hexadecimal.*
Memory address numbers can be large values, and thus may be written more compactly in hexadecimal than in decimal. For example, 1,000,000 in decimal is f4240 in hexadecimal.
Converting between hexadecimal and decimal is explained next in the upcoming section, "Converting Between Decimal and Binary or Hexadecimal."

Bits and Bytes

While people live at street addresses, what is stored at each memory address is a *byte.* Don't worry, I have not misspelled Dracula's favorite pastime.

As discussed in Chapter 1, early computers essentially were a series of switches, 1 representing on, 0 representing off. In computer terminology, a *bit* is either a 1 or a 0.

However, while a computer may think in bits, it cannot process information as small as a single bit. Eight bits, or one *byte,* is the smallest unit of information that a computer can process.

Accordingly, each address may store up to one byte of information, represented by a sequence of up to eight ones and zeroes. Thus, just as a street address may be used to locate the persons who live there, a memory address can be used to locate the one byte of information that is stored there. Figure 2-2 shows a sequence of memory addresses, each with a value.

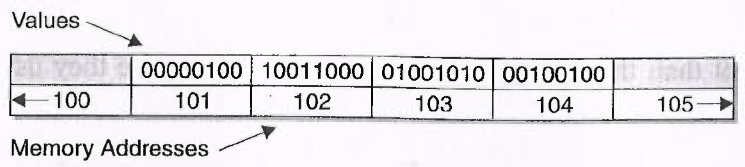


Figure 2-2 A sequence of memory addresses, each with a byte value

Binary Numbering System

The information stored at a memory address, a series of ones and zeroes, probably has little meaning to most of us. However, to a computer, a sequence of ones and zeroes is quite meaningful.

For example, to my computer, I was born in the year 11110100000. Before you tell me that's impossible, I will tell you I was born in the year 1952. How could I have been born both in the year 11110100000 and in the year 1952?

The numbers with which we usually work are decimal, or base 10. Each number in decimal is represented by a digit between 0 and 9. 1952 is a decimal number.

The sequence of ones and zeroes in a byte also is a number, though it may not look like any number you have ever seen. My birth year, expressed as the number 11110100000, is binary, or base 2. Each number in binary is represented by a digit that is either 0 or 1.

The reason both decimal and binary numbers are involved in computer programming is because both humans and computers are involved. While humans think in decimal numbers, computers "think" in binary numbers.

Converting Between Decimal and Binary or Hexadecimal

You can write computer programs without knowing how to convert between binary and decimal numbers. However, knowing how to do so is not difficult and may help your understanding of what happens behind the scenes. If you are interested, read on!

Converting a number from binary to decimal is simple. Going from right to left, the rightmost binary digit is multiplied by 2^0, or 1, the second binary digit from the right is multiplied by 2^1, or 2, the third binary digit from the right is multiplied by 2^2, or 4, and so on, through all of the binary digits. The results of each multiplication are added, and the result is the decimal equivalent of the binary number. Table 2-1 shows this calculation for the binary equivalents of the numbers 1 through 5 in decimal.

Binary	Calculation	Decimal
0	$0 \times 2^0 = 0 \times 1 =$	0
1	$1 \times 2^0 = 1 \times 1 =$	1
10	$(0 \times 2^0) + (1 \times 2^1) = 0 + 2$	2
11	$(1 \times 2^0) + (1 \times 2^1) = 1 + 2$	3
100	$(0 \times 2^0) + (0 \times 2^1) + (1 \times 2^2) = 0 + 0 + 4$	4
101	$(1 \times 2^0) + (0 \times 2^1) + (1 \times 2^2) = 1 + 0 + 4$	5

Table 2-1 Binary Equivalents of the Numbers 1 Through 5 in Decimal

Converting a number from decimal to binary is almost as easy. Let's use 5 in decimal as an example.

1. You find the largest power of 2 that can be divided into 5 with a quotient of 1. The answer is 2^2, or 4.
2. Remember when converting from binary to decimal, the rightmost binary digit is multiplied by 2^0, or 1, the second binary digit from the right is multiplied by 2^1, the third binary digit from the right is multiplied by 2^2, and so on. Since the exponent is 2, a binary 1 goes into the third binary digit from the right, so the binary number now is 1??, the ? representing each binary digit we still need to calculate.
3. When you divide 5 by 4, the remainder is 1. You next try to divide 1 by the next lowest power of 2, 2^1, or 2. The quotient is 0, so a binary 0 goes into the second binary digit from the right. The binary number now is 10?.
4. When you divide 1 by 2, the remainder is still 1. You next try to divide 1 by the next lowest power of 2, 2^0, or 1. The quotient is 1, so a binary 1 goes into the rightmost binary digit. The binary number now is 101, and we're done.

You also can use the same techniques for converting between hexadecimal and decimal. When converting from hexadecimal to decimal, multiply each hexadecimal digit (converting a to 10, b to 11, and so on) by the appropriate power of 16. For example, 5c in hexadecimal is $(12 \times 16^0) + (5 \times 16^1)$, which is 12 + 80 or 92.

Conversely, when converting from decimal to hexadecimal, the highest power of 16 that can be divided into 92 is 16^1, or 16. The quotient is 5, which goes into the second digit to the right. The remainder is 12, which is c in hexadecimal. This goes into the rightmost digit, resulting in the hexadecimal number 5c.

Data Types

The ones and zeroes that may be stored at a memory address may represent text, such as my name, Jeff Kent. These ones and zeroes instead may represent a whole number, such as my height in inches, 72, or a number with digits to the right of the decimal point, such as my GPA in high school, which I'll say was 3.75 (I honestly don't remember, it was too long ago). Alternatively, the ones and zeroes may represent either true or false, such as whether I am a U.S. citizen.

Data comes in many forms, and is generally either numeric or textual. Additionally, some numeric data uses whole numbers, such as 6, 0, or –7, while other numeric data uses floating-point numbers, such as .6, 7.3, and –6.1.

There are different data types for each of the many forms of data. The data type you choose will affect not only the form in which the data is stored, but also the amount of memory required to store the data. Let's now take a look at these different data types.

Whole Number Data Types

We deal with whole numbers all the time. Think of the answers to questions such as how many cars are in the parking lot, how many classes are you taking, or how many brothers and sisters do you have? Each answer involves a number, with no need to express any value to the right of the decimal point. After all, who has 3.71 brothers and sisters?

Often, you don't need a large whole number. What unfortunate student would be taking 754,361 classes at one time? However, sometimes the whole number needs to be large. For example, if you are studying astronomy, the moon is approximately 240,000 miles from Earth. Indeed, sometimes the whole number may need to be very, very large. Pluto's minimum distance from the Earth is about 2.7 billion miles.

Many times, the whole number won't be negative. No matter how badly you do on a test, chances are you won't score below zero points. However, some whole numbers may be below zero, such as the temperature at the North Pole.

Because of the different needs whole numbers may have to meet, there are several different whole number data types (shown in Table 2-2). The listed sizes and ranges are typical, but may vary depending on the compiler and operating system. In the *sizeof* operator project later in this chapter, you will determine through code the size of different data types on your compiler and operating system.

Data Type	Size (in Bytes)	Range
short	2	–32,768 to 32,767
unsigned short	2	0 to 65,365
int	4	–2,147,483,648 to 2,147,483,647
unsigned int	4	0 to 4,294,987,295
long	4	–2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,987,295

Table 2-2 Whole Number Data Types, Sizes, and Ranges

NOTE: You may be wondering about the purpose of the long data type, since its size and range is the same as an int in Table 2-2. However, as noted just before that table, the actual size, and, therefore, range of a particular data type varies depending on the compiler and operating system. On some combinations of compilers and operating systems, short may be 1 byte, int may be 2 bytes, and long may be 4 bytes.

Beginning programmers sometimes see information like that shown in Table 2-2 and panic that they can't possibly memorize all of it. The good news is you don't have to. To be sure, some memorization is necessary for almost any task. However, since there really is too much information to memorize, programmers frequently resort to online help or reference books. Believe me, I do.

Far more important to a programmer than rote memorization is to understand how and why a program works as it does. Therefore, this section will go into some detail as to how data types work. Some arithmetic necessarily is involved, but it is not difficult, and if you follow the arithmetic, you will have a good understanding of data types that will help you in your programming in the following chapters.

Unsigned vs. Signed Data Type

Table 2-2 lists three data types: short, int, and long. Each of these three data types has either the word unsigned in front of it or nothing at all—as in unsigned short and short.

Unsigned means the number is always zero or positive, never negative. *Signed* means the number may be negative or positive (or zero). If you don't specify signed or unsigned, the data type is presumed to be signed. Thus, signed short and short are the same.

Since an unsigned data type means its value is always 0 or positive, never negative, in Table 2-2 the smallest value of an unsigned short is therefore zero; an unsigned short cannot be negative. By contrast, the smallest value of a short is –32767, since a signed data type may be negative, positive, or zero.

Size

Each of the whole number data types listed in Table 2-2 has a size. Indeed, all C++ data types have a size. However, unlike people, the size of a data type is not expressed in inches or in pounds (a sore subject for me), but in bytes.

Since a byte is the smallest unit of information that a computer can process, no data type may be smaller than one byte. Most data types are larger than one byte; all the whole number data types listed in Table 2-2 are. However, regardless of the size, the number of bytes is always a whole number. You cannot have a data type whose size is 3.5 bytes because .5 bytes, or 4 bits, is too small for the computer to process.

Generally, the number of bytes for a data type is the result of a power of 2 since computers use a binary number system. Thus, typical data type sizes are 1 byte (2^0), 2 bytes (2^1), four bytes (2^2), or eight bytes (2^3).

The size of a data type matters in two related respects: (1) the *range* of different values that the data type may represent and (2) the amount of memory required to store the data type.

Range

Range means the highest and lowest value that may be represented by a given data type. For example, the range of the unsigned short data type is 0 to 65,365. These lowest and highest values are not arbitrary, but instead can be calculated.

The number of different values that a data type can represent is 2^n, *n* being the number of bits in the data type. The size of a short data type is 2 bytes, or 16 bits. Therefore, the number of different whole numbers that the short data type can represent is 2^{16}, which is 65,356.

However, the highest value that an unsigned short can represent is 65,355, not 65,356, because the unsigned short data type starts at 0, not 1. Therefore, the highest number that an unsigned data type may represent is $2^n - 1$; *n* again being the number of bits in the data type, and the minus 1 being used because we are starting at 0, not 1.

Signed data types involve an additional issue. Since the range of a signed data type includes negative numbers, there needs to be a way of determining if a number is positive or negative. We determine if a decimal number is positive or negative by looking to see if the number is preceded by a negative sign (–). However, a bit can be only 1 or 0; there is no option for a negative sign in a binary number.

There are several different explanations in computer science for the representation of negative numbers, such as *signed magnitude, one's complement,* and *two's complement.* However, we don't need to get into the complexities of these explanations.

For example, a signed short data type, like an unsigned short data type, can represent 2^{16} or 65,356 different numbers. However, with a signed data type, these different numbers must be split evenly between those starting at zero and going up, and those starting at zero and going down. To do this, the two ranges would be 0 to 32,767 and –1 to –32,768. This can be confirmed by Table 2-2, which shows the range of a signed data type as –32,768 to 32,767.

Another way of explaining the high and low numbers of the range of the signed short data type is that one of the bits is used to store the sign, positive or negative. That leaves 15 bits. The highest number in the range is $2^{15} - 1$, or 32,767; the minus 1 being used because we are starting at 0, not 1. The lowest number in the range is $-(2^{15})$, or –32,768; there's no minus 1 because we are starting at –1, not 0.

Storage

In binary, 65365 as an unsigned short is represented by sixteen ones: 1111111111111111. You cannot fit 16 bits into a single memory address. A memory address can hold only 8 bits, or a byte. How then can you store this value in memory?

The answer is you need two memory addresses to store 65365 in decimal. This provides two bytes of storage, sufficient to store this value. This is why the short data type requires 2 bytes of storage. Figure 2-3 shows how this value would be stored as a short data type.

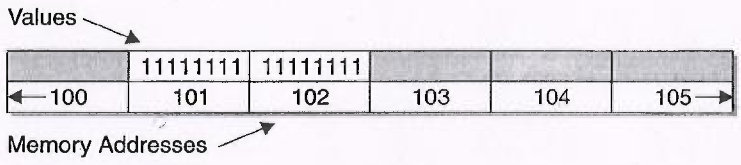


Figure 2-3 Storage in memory of 65365 in decimal as an unsigned short data type

The int data type requires 4 bytes of storage. Figure 2-4 shows how 65365 in decimal would be stored as an unsigned int data type.

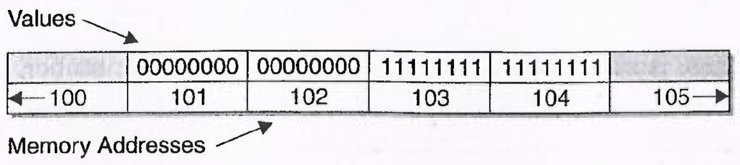


Figure 2-4 Storage in memory of 65365 in decimal as an unsigned int data type

You may legitimately wonder why 65365 in decimal as an unsigned int data type requires four bytes of storage when 65365 in decimal as an unsigned short data type requires only two bytes of storage. In other words, if you specify int instead of short as the data type, four bytes of storage will be reserved, even if you could store the number in less bytes. The reason is that it is not known, when memory is reserved, what value will be stored there. Additionally, the value could change. Accordingly, enough bytes of storage are reserved for the maximum possible value of that data type.

Why Use a Smaller Size Data Type?

Given that an int can store a far wider range of numbers than a short, you also may be wondering why you ever would use a short rather than an int. The answer is that the wider range of an int comes at a price; it requires twice as much RAM as a short—four instead of two bytes.

However, computers these days come with hundreds of megabytes of RAM, each megabyte being 1,048,576 bytes; you still may wonder why you should care about two measly extra bytes. If it *was* just 2 extra bytes, you wouldn't care. However, if you are writing a program for an insurance company that has one million customers, you won't be talking about 2 extra bytes, but instead 2 *million* extra bytes. Therefore, you should not just reflexively choose the largest data type.

All this said, as a general rule, of the six whole number data types, you most often will use int. However, it is good to know about the other choices.

Floating-Point Data Types

I was nearsighted my entire adult life until I had lasik surgery on my eyes. In this surgery, the eye surgeon programs information that the laser used to reshape my eyeball by shaving off very thin slices of my cornea, measuring only thousandths of an inch, in certain areas of my eyeball, leaving untouched other areas, again only thousandths of an inch away.

Can you imagine my reaction if the eye surgeon had told me his philosophy was "close enough for government work," so he was using only whole numbers, ignoring any values to the right of the decimal point? You next would have seen my silhouette through the wall after I ran through it to escape. (Since I still go to my eye surgeon, who, by the way, earned his way through college as a computer programmer, and it is not in my best interest to get on his bad side, let me hasten to add that he was very precise and the surgery was successful.)

Whole numbers work fine for certain information where fractions don't apply. For example, who would say they have 2 ¾ children? Whole numbers also work fine for certain information where fractions do apply but are not important. For example, it would be sufficient normally to say the location is 98 miles away; precision such as 98.177 miles usually is not necessary.

However, other times fractions, expressed as numbers to the right of the decimal point, are very important. My lasik surgery is an extreme example, but there are many other more common ones. If you had a 3.9 GPA, you probably would not want the school to just forget about the .9 and say your GPA was 3. Similarly, a bank that kept track of dollars but not cents with deposits and withdrawals would, with potentially millions of transactions a day, soon have very inaccurate information as to how much money it has, and its depositors have.

Accordingly, there are floating-point data types that you can use when a value to the right of the decimal point is important. The term *floating point* comes from the fact that there is no fixed number of digits before and after the decimal point; that is, the decimal point can float. Floating-point numbers also are sometimes referred to as real numbers.

Table 2-3 lists each of the floating-point number data types. As with the whole number data types, the listed sizes and ranges are typical, but may vary depending on the compiler and operating system.

Data Type	**Size (in Bytes)**	**Range (in E notation)**
float	4	±3.4E-38 to ±3.4E38
double	8	±1.7E-308 to ±1.7E308
long double	10	±3.4E-4932 to ±3.4E4932

Table 2-3 Floating-point Number Data Types, Sizes, and Ranges

NOTE: *The size of a long double on many combinations of compilers and operating systems may be 8 bytes, not 10.*

Scientific and E Notations

The range column in Table 2-3 may not look like any number you have ever seen before. That is because these are not usual decimal numbers, but instead numbers expressed in *E notation*, the letter *E* standing for exponent.

The float data types can store very large numbers, such as (in decimal) 1000000000000000000000000000000000000, which could be a distance across the universe. The float data types also can store very small numbers, such as .0000000000000000000000000000000000001, which could be the diameter of a subatomic particle.

Rather than having digits running across the page, the number can be expressed more compactly. One way is with *scientific notation,* another is with E notation. Table 2-4 shows how certain floating-point numbers are represented in both notations.

Decimal Notation	**Scientific Notation**	**E Notation**
123.45	1.2345×10^{2}	1.2345E2
0.0051	5.1×10^{-3}	5.1E-3
1,200,000,000	1.2×10^{9}	1.2E9

Table 2-4 Scientific and E Notation Representations of Floating Point Values

In scientific notation, the number before the multiplication operator, called the *mantissa*, always is expressed as having a single digit to the left of the decimal point, and as many digits as necessary to the right side of the decimal point to express the number. The number after the multiplication operator is a power of 10, which may be positive for very large numbers or negative for very small fractions. The value of the expression is the mantissa multiplied by the power of 10.

E notation is very similar to scientific notation. The only difference is the multiplication operator, followed by 10 and an exponent, is replaced by an E followed by the exponent.

Storage of Floating-Point Numbers

Since only ones and zeroes can be stored in memory, complex codes, well beyond the scope of this book, are required to store floating-point numbers. Even with complex codes, a computer can only approximately represent many floating-point values. Indeed, in certain programs the programmer has to take care to ensure that small discrepancies in each of a number of approximations don't accumulate to the point where the final result is wrong.

NOTE: Because mathematics with floating-point numbers requires a great deal of computing power, many CPUs come with a chip specialized for performing floating-point arithmetic. These chips often are referred to as math coprocessors.

Text Data Types

There are two text data types. The first is *char,* which stands for character. It usually is 1 byte, and can represent any single character, including a letter, a digit, a punctuation mark, or a space.

The second text data type is *string.* The string data type may store a number of characters, including this sentence, or paragraph, or page. The number of bytes required depends on the number of characters involved.

NOTE: Unlike char and the other data types we have discussed, the string type is not a data type built into C++. Instead, it is defined in the standard library file string, which therefore must be included with an include directive (#include <string>) to use the string data type. Chapter 1 covers the include directive, which in the "Hello World!" program was #include <iostream>.

Storage of Character Values

There is a reason why the size of a character data type usually is 1 byte.

ANSI (American National Standards Institute) and ASCII (American Standards Committee for Information Interchange) adopted for the English language a set of 256 characters, which includes all alphabetical characters (upper- and lowercase), digits and punctuation marks, and even characters used in graphics and line drawing. Each of these 256 different characters is represented by a number between 0 and 255 that it corresponds to. Table 2-5 lists the ASCII values of commonly used characters.

Each of the 256 different values can be represented by different combinations of 8 bits, or one byte. This is true because 2^8 equals 256. Thus, 00000000 is equal to 0, the smallest ASCII value, and 11111111 is equal to 255, the largest ASCII value.

For example, the letter J has the ASCII code 74. The binary equivalent of 74 is 1001010. Thus, 1001010 at a memory address could indicate the letter J.

Characters	Values	Comments
0 through 9	48–57	0 is 48, 9 is 57
A through Z	65–90	A is 65, Z is 90
a through z	97–122	a is 97, z is 122

Table 2-5 ASCII Values of Commonly Used Characters

Note: 1001010 also could indicate the number 74; you wouldn't know which value was being represented unless you knew the data type associated with that memory address. In the next chapter, you will learn about variables, which enable you to associate a particular data type with a specific memory address.

Storage of Strings

The amount of memory required for a string depends on the number of characters in the string. However, each memory address set aside for the string would store one character of the string.

The bool Data Type

There is one more data type, *bool.* This data type has only two possible values, true and false, and its size usually is one byte. The term "bool" is a shortening of Boolean, which is usually used in connection with Boolean Algebra, named after the British mathematician, George Boole.

The *bool* data type is mentioned separately since it does not neatly fit into either the number or text categories. It could be regarded as a numeric data type in that zero is seen as false, and one (or any other non-zero number) as true. While it may not seem intuitive why zero would be false and one would be true, remember that computers essentially store information in switches, where 1 is on, and 0 is off.

Project: Determining the Size of Data Types

As discussed in the previous Data Types section, the size of each data type depends on the compiler and operating system you are using. In this project, you will find out the size of each data type on your system by using the *sizeof* operator.

The sizeof Operator

The *sizeof* operator is followed by parentheses, in which you place a data type. It returns the size in bytes of that data type.

For example, on my computer, the expression sizeof(int) returns 4. This means that on my compiler and operating system, the size of an int data type is 4 bytes.

Changing the Source File of Your Project

Try creating and running the next program using the steps you followed in Chapter 1 to create the "Hello World!" program. While you could start a new project, in this example, you will reuse the project you used in Chapter 1. It is good to know both how to create a new project and how to reuse an existing one.

1. Start Visual C++.
2. Use the File | Open Solution menu command to display the Open Solution dialog box shown in Figure 2-5.

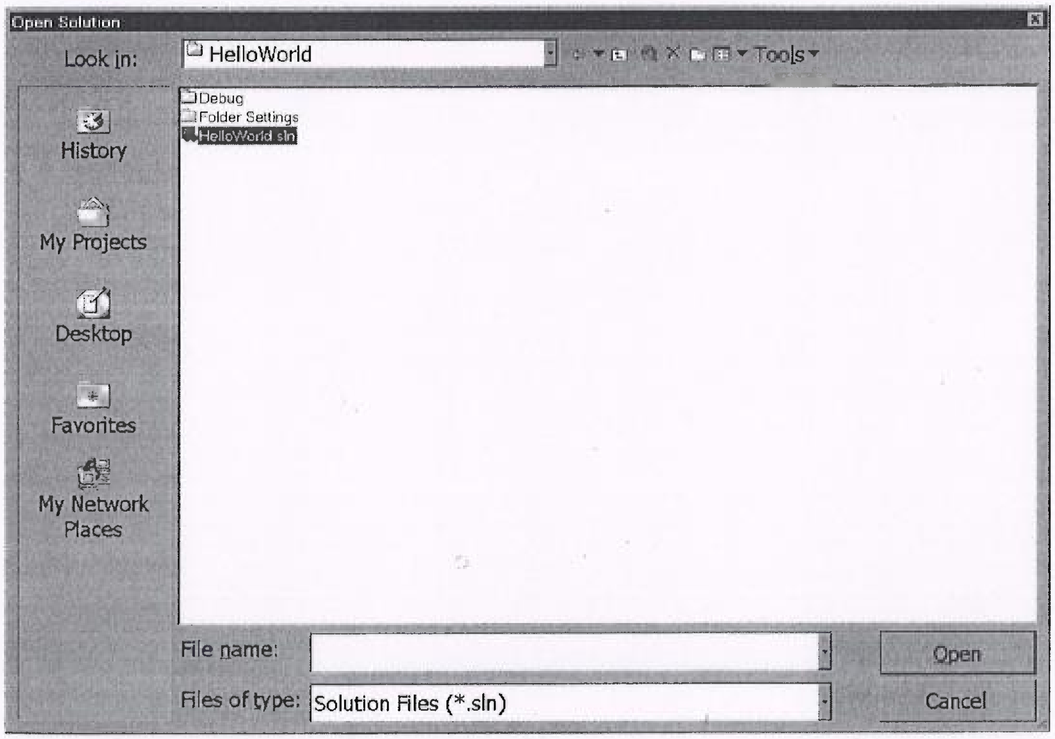


Figure 2-5 Opening the Existing Solution

3. Navigate to the folder where you saved the project (C:\temp\helloworld on my computer) and find the solution file. It has the extension .sln, which stands for solution. The solution file is helloworld.sln in Figure 2-5.
4. Open the solution file. This should open your project.
5. Display Solution Explorer using the View | Solution Explorer menu command, and then click the Source Files folder to show the hello.cpp file, as depicted in Figure 2-6.

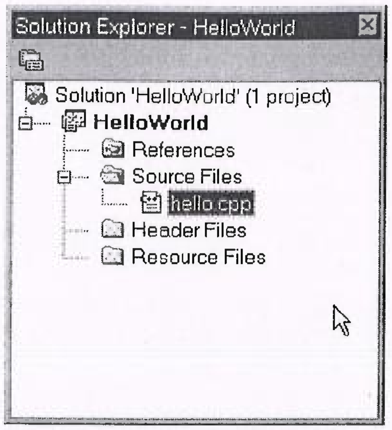


Figure 2-6 Showing the Existing Source File in Solution Explorer

6. Right-click the hello.cpp file and choose Remove from the shortcut menu (shown in Figure 2-7). Don't worry, this will not delete the file, but instead simply remove it from the project. You still will be able to use it later if you wish.

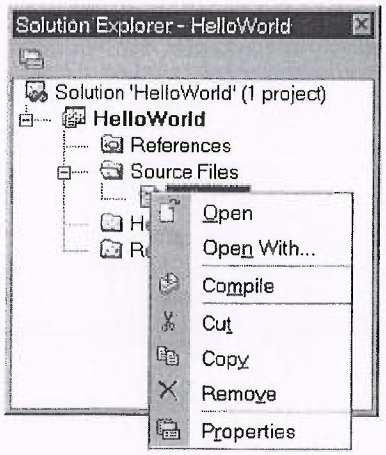
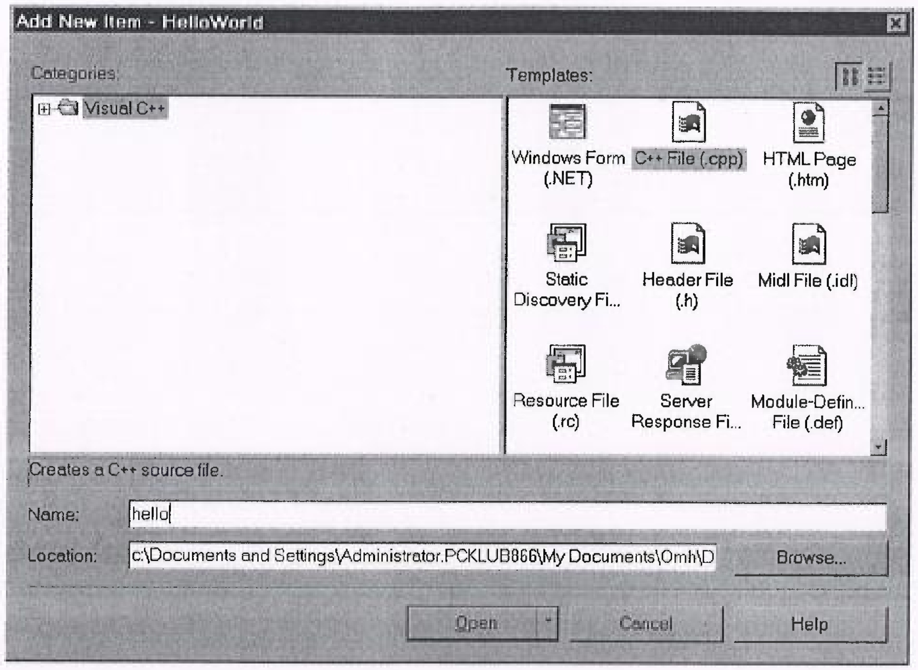


Figure 2-7 Remove option on Shortcut Menu

7. Right-click the Source Files folder and choose Add New Item from the shortcut menu. This will display the Add New Item dialog box, shown in Figure 2-8.



Creates a C++ source file.

Name: hello

Location: c:\Documents and Settings\Administrator.PCKLUB866\My Documents\Omh\D

Figure 2-8 Adding a New Source File to your Project

8. Don't change the Location field, which holds the subfolder in which the project files are stored. Type the name of the new source file in the Name field, such as **sizeof.cpp**.
9. When you are done, click the Open button. Figure 2-9 shows the new sizeof.cpp file in Solution Explorer.

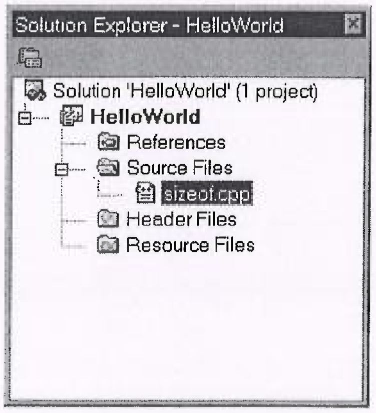


Figure 2-9 Solution Explorer showing the new .cpp file

Double-click sizeof.cpp in Solution Explorer to display the sizeof.cpp file in the code editing window. At this point, the sizeof.cpp is blank. In the next section, you will add code.

Code and Output

Write the following code in the source file you have created. I will explain the code in the following sections.

```
#include <iostream>
using namespace std;
int main(void)
{
   cout << "Size of short is " << sizeof(short) << "\n";
   cout << "Size of int is " << sizeof(int) << "\n";
   cout << "Size of long is " << sizeof(long) << "\n";
   cout << "Size of float is " << sizeof(float) << "\n";
   cout << "Size of double is " << sizeof(double) << "\n";
   cout << "Size of long double is
         " << sizeof(long double) << "\n";
   cout << "Size of char is " << sizeof(char) << "\n";
   cout << "Size of bool is " << sizeof(bool) << "\n";
return 0;
}
```

Next, build and run the project, following the same steps you did for the "Hello World!" Project in Chapter 1. The resulting output on my computer is

```
Size of short is 2
Size of int is 4
Size of long is 4
Size of float is 4
Size of double is 8
Size of long double is 8
Size of char is 1
Size of bool is 1
```

NOTE: *The numbers displayed on your computer may be different, because the size of a data type depends on the particular compiler and operating system you are using, and yours may not be the same as mine.*

Expressions

The line of code

```
    cout << "Size of int is " << sizeof(int) << "\n";
```

displays the following output:

```
Size of int is 4
```

In essence, the code sizeof(int) is replaced by 4 in the output.

The code sizeof(int) is called an *expression.* An expression is a code statement that has a value, usually a value that has to be evaluated when the program runs. An example of an expression is 4 + 4, which has a value, 8, that would be evaluated when the program runs.

When the code runs, the expression sizeof(int) is evaluated as having the value 4, which then is outputted.

By contrast, the portion of the statement within double quotes, "Size of int is ," is outputted literally as "Size of int is 4." There is no need for an evaluation. Instead, this is considered a *literal string.* The term string refers to the data type, a series of characters, and the term literal refers to the fact that the string is outputted literally, without evaluation. The string "Hello World!" in the cout statement in Chapter 1 also was a literal string.

Outputting an Expression

The expression sizeof(int) is separated by the stream insertion operator (<<) from the literal string "Size of int is ." If the code statement instead were

```
    cout << "Size of int is sizeof(int)\n";
```

then the output would be quite different:

```
Size of int is sizeof(int)
```

The reason is sizeof(int), being encased inside the double quotes, would be treated as a literal string, not an expression, and therefore would not be evaluated, but instead displayed as is.

Since "Size of int is" is a literal string and sizeof(int) is an expression, they need to be differentiated before being inserted into the output stream. This differentiation is done by placing a stream insertion operator between the literal string and the expression.

Note: The string "Size of int is" ends with a space between "is" and the following 4. Without that space, the output would be "Size of int is4." You, as the programmer, have the responsibility to ensure proper spacing; C++ won't do it for you.

Escape Sequences

The string "\n" following the expression sizeof(int) is also a literal string, so it, too, is separated by a stream insertion operator from the sizeof(int) expression. However, "\n" is a special type of string called an *escape sequence.*

C++ has many escape sequences, though this may be the commonest one. This particular escape sequence causes the cursor to go to the next line for further printing. Without it, all the output would be on one line.

The "\n" in a string is not displayed literally by cout even though it is encased in double quotes. The reason is that the backslash signals cout that this is an escape sequence.

Table 2-6 shows some of the most common escape sequences.

Escape Sequence	Name	What It does
\a	Alarm	Causes the computer to beep
\n	newline	Causes the cursor to go to the next line
\t	Tab	Causes the cursor to go to the next tab stop
\\	Backslash	Causes a backslash to be printed
\'	Single quote	Causes a single quote to be printed
\"	Double quote	Causes a single quote to be printed

Table 2-6 Common Escape Sequences

Summary

A computer program's instructions and data have to be in the computer's memory for the program to work. There are three principal memory locations on your computer: the central processing unit (CPU), random access memory (RAM), and persistent storage. Computer programs usually use RAM to store instructions and data.

Instructions and data are stored at addresses, represented by a sequential series of numbers. A computer stores information in a series of ones and zeroes. Each one or zero is a bit. However, a computer cannot process information as small as a single bit. Eight bits, or one *byte,* is the smallest unit of information that a computer can process. Therefore, each address stores one byte of information.

Some information is numeric; other data is textual. Each type of information is referred to as a data type. The principal data type categories are whole numbers, floating-point numbers, and text. However, all data types have in common a characteristic of size, which is the number of bytes required to store information of that data type. A data type's size also determines its range, which is the highest and lowest number that can be stored by that data type.

The size of a data type varies depending on the compiler and operating system. You may use the sizeof operator to determine the size of a data type on your particular system.

Quiz

1. From which of the following types of memory can the CPU most quickly access instructions or data: cache memory, RAM, or persistent storage?
2. Which of the following types of memory is not temporary: cache memory, RAM, or persistent storage?
3. What is the amount of information that may be stored at a particular memory address?
4. Is the size of a data type always the same no matter which computer you may be working on?
5. What is meant by the range of a data type?
6. What is the difference between an unsigned and signed data type?
7. What decimal number is represented by 5.1E-3 in E notation?
8. What is an ASCII value?
9. What does the sizeof operator do?
10. What is a literal string?
11. What is an expression?

CHAPTER 3

Variables

Recently, while in a crowded room, someone yelled "Hey, you!" I and a number of other people looked up, because none of us could tell to whom the speaker was referring. Had the speaker instead yelled "Hey, Jeff Kent!," I would have known he was calling me (unless of course there happened to be another Jeff Kent in the room).

We use names to refer to each other. Similarly, when you need to refer in code to a particular item of information among perhaps thousands of items of information, you do so by referring to the name of that information item.

You name information by creating a variable. A variable not only gives you a way of referring later to particular information, but also reserves the amount of memory necessary to store that information. This chapter will show you how to create variables, store information in them, and retrieve information from them.

Declaring Variables

You learned in Chapter 2 that the information a program uses while it is running first needs to be stored in memory. You need to reserve memory before you can store information there. You reserve memory by *declaring* a variable.

Declaring a variable not only reserves memory, but also gives you a convenient way of referring to that reserved memory when you need to do so in your program. You also learned in Chapter 2 that memory addresses have hexadecimal values such as 0012FED4. These values are hard to remember. It is much easier to remember information that, for example, relates to a test score by the name testScore. By declaring a variable, you can refer to the reserved memory by the variable's name, which is much easier to remember and identify with the stored information than is the hexadecimal address.

While declaring a variable is relatively simple, requiring only one line of code, much is happening behind the scenes. The program at the end of this section will show you how to determine the address and size of the memory reserved by declaring a variable.

Syntax of Declaring Variables

You have to *declare* a variable before you can use it. Declaring a variable involves the following syntax:

```
[data type] [variable name] ;
```

The data type may be any of the ones discussed in Chapter 2, including int, float, bool, char, or string. The data type tells the computer how much memory to reserve. As you learned in Chapter 2, different data types have different sizes in bytes. If you specify a data type with a size (on your compiler and operating system) of 4 bytes, then the computer will reserve 4 bytes of memory.

You choose the variable name; how you name a variable is discussed later in the section "Naming the Variable." The name is an alias by which you can refer in code to the area of reserved memory. Thus, when you name a variable that relates to a test score *testScore,* you can refer in code to the reserved memory by the name *testScore* instead of by a hexadecimal value such as 0012FED4.

Finally, the variable declaration ends with a semicolon. The semicolon tells the compiler that the statement has ended. You can declare a variable either within a function, such as main, or above all functions, just below any include directives. Since for now our programs have only one function, main, we will declare all variables within main. When our programs involve more than one function, we will revisit the issue of where to declare variables.

The following statement declares in main an integer variable named *testScore.*

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

Note: *Unlike the code in Chapters 1 and 2, there is no include directive such as #include <iostream> in this code because this code does not use cout or another function defined in a standard library file.*

You will receive a compiler error if you refer to a variable before declaring it. In the following code, the reference to *testScore* will cause the compiler error "undeclared identifier."

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

This compiler error will occur even though the variable is declared in the very next statement. The reason is that the compiler reads the code from top to bottom, so when it reaches the first reference to *testScore,* it has not seen the variable declaration.

This "undeclared identifier" compiler error is similar to the one in the "Hello World!" project in Chapter 1 when we (deliberately) misspelled cout as Cout. Since *testScore* is not a name built into C++, like main and int, the compiler does not recognize it. When you declare a variable, then the compiler recognizes further references to the variable name as referring to the variable that you declared.

Declaring Multiple Variables of the Same Data Type

If you have several variables of the same data type, you could declare each variable in a separate statement.

```
    int testScore;
    int myWeight;
    int myHeight;
```

However, if the variables are of the same data type, you don't need to declare each variable in a separate statement. Instead, you can declare them all in one statement, separated by commas. The following one statement declares all three integer variables:

```
    int testScore, myWeight, myHeight;
```

The data type int appears only once, even though three variables are declared. The reason is that the data type qualifies all three variables, since they appear in the same statement as the data type.

However, the variables must all be of the same data type to be declared in the same statement. You cannot declare an int variable and a float variable in the same statement. Instead, the int and float variables would have to be declared in separate statements.

```
int testScore;
float myGPA;
```

Naming the Variable

Variables, like people, have names, which are used to identify the variable so you can refer to it in code. There are only a few limitations on how you can name a variable.

- The variable name cannot begin with any character other than a letter of the alphabet (A–Z or a–z) or an underscore (_). Secret agents may be named 007, but not variables. However, the second and following characters of the variable name may be digits, letters, or underscores.
- The variable name cannot contain embedded spaces, such as *My Variable,* or punctuation marks other than the underscore character (_).
- The variable name cannot be the same as a word reserved by C++, such as main or int.
- The variable name cannot have the same name as the name of another variable declared in the same scope. Scope is an issue that will be discussed in Chapter 8. For present purposes, this rule means you cannot declare two variables in main with the same name.

Besides these limitations, you can name a variable pretty much whatever you want. However, it is a good idea to give your variables names that are meaningful. If you name your variables *var1, var2, var3,* and so on, up through *var17,* you may find it difficult to later remember the difference between *var8* and *var9.* And if you find it difficult, imagine how difficult it would be for a fellow programmer, who didn't even write the code, to figure out the difference.

In order to preserve your sanity, or possibly your life in the case of enraged fellow programmers, I recommend you use a variable name that is descriptive of the purpose of the variable. For example, *testScore* is descriptive of a variable that represents a test score.

The variable name *testScore* is a combination of two names: test and score. You can't have a variable name with embedded spaces such as *test score.* Therefore, the two words are put together, and differentiated by capitalizing the first letter of the second word. By the convention I use, the first letter of a variable name is not capitalized.

Naming Conventions

A naming convention is simply a consistent method of naming variables. There are a number of naming conventions. In addition to the one I described earlier, another naming convention is to name a variable with a prefix, usually all lowercase and consisting of three letters, that indicate its data type, followed by a word with its first letter capitalized, that suggests its purpose. Some examples:

- *intScore* Integer variable representing a score, such as on a test.
- *strName* String variable representing a name, such as a person's name.
- *blnResident* Boolean variable, representing whether or not someone is a resident.

It is not particularly important which naming convention you use. What is important is that you use one and stick to it.

The Address Operator

Declaring a variable reserves memory. You can use the *address operator* (&) to learn the address of this reserved memory. The syntax is

```
&[variable name]
```

For example, the following code outputs 0012FED4 on my computer. However, the particular memory address for *testScore* on your computer may be different than 0012FED4. Indeed, if I run this program again some time later, the particular memory address for *testScore* on my computer may be different than 0012FED4.

```
#include <iostream>
using namespace std;
int main(void)
{
   int testScore;
   cout << &testScore;
   return 0;
}
```

The address 0012FED4 is a hexadecimal (Base 16) number. As discussed in Chapter 2, memory addresses usually are expressed as a hexadecimal number.

The operating system, not the programmer, chooses the address at which to store a variable. The particular address chosen by the operating system depends on the data type of the variable, how much memory already has been reserved, and other factors.

You really do not need to be concerned about which address the operating system chose since your code will refer to the variable by its name, not its address. However, as you will learn in Chapter 11 when we discuss pointers, the address operator can be quite useful.

Using the Address and sizeof Operators with Variables

The amount of memory reserved depends on a variable's data type. As you learned in Chapter 2, different data types have different sizes.

In Chapter 2, you used the sizeof operator to learn the size (on your compiler and operating system) of different data types. You also can use the sizeof operator to determine the size (again, on your compiler and operating system) of different variables.

The syntax for using the sizeof operator to determine the size of a variable is almost the same as the syntax for using the sizeof operator to determine the size of a data type. The only difference is that the parentheses following the sizeof operator refers to a variable name rather than a data type name.

The following code outputs the address and size of two variables:

```
#include <iostream>
using namespace std;
int main(void)
{
   short testScore;
   float myGPA;
   cout << "The address of testScore is "
           << &testScore << "\n";
   cout << "The size of testScore is "
           << sizeof(testScore) << "\n";
   cout << "The address of myGPA is " << &myGPA << "\n";
   cout << "The size of myGPA is "
           << sizeof(myGPA) << "\n";
   return 0;
}
```

The output when I ran this program (yours may be different) is

```
The address of testScore is 0012FED4
The size of testScore is 2
The address of myGPA is 0012FEC8
The size of myGPA is 4
```

Figure 3-1 shows how memory is reserved for the two variables. Due to the different size of the variables, the short variable, *testScore,* takes up two bytes of memory, and the float variable, *myGPA,* takes up four bytes of memory.

float myGPA					short testScore	
0012FEC8	0012FEC9	0012FECA	0012FEFB	....	0012FED4	0012FED5

Figure 3-1 Memory reserved for declared variables

As Figure 3-1 depicts, the addresses of the two variables are near each other. The operating system often attempts to do this. However, this is not always possible, depending on factors such as the size of the variables and memory already reserved. There is no guarantee that two variables will even be near each other in memory.

In Figure 3-1, the value for both memory addresses is unknown. That is because we have not yet specified the values to be stored in those memory locations. The next section shows you how to do this.

Assigning Values to Variables

The purpose of a variable is to store information. Therefore, after you have created a variable, the next logical step is to specify the information that the variable will store. This is called *assigning* a value to a variable.

A variable can be assigned a value supplied by the programmer in code. A variable also can be assigned a value by the user, usually via the keyboard, when the program is running.

You may use the assignment operator, which is discussed in the next section, to specify the value to be stored in a variable. You use the cin object (discussed in the upcoming section "Using the cin Object") after the assignment operator, to obtain the user's input, usually from the keyboard, and then store that input in a variable.

Assignment Operator

You use the assignment operator to assign a value to a variable. The syntax is

```
[variable name] = [value];
```

The assignment operator looks like the equal sign. However, in C++ the = sign is not used to test for equality; it is used for assignment. As you will learn in Chapter 5, in C++ the equal sign is ==, also called the equality operator.

The variable must be declared either before, or at the same time, you assign it a value, not afterwards. In the following example, the first statement declares the variable, and the second statement assigns a value to that variable:

```
int testScore;
testScore = 95;
```

The next example concerns *initialization,* which is when you assign a value to a variable as part of the same statement that declares that variable:

```
int testScore = 95;
```

However, the variable cannot be declared after you assign it a value. The following code will cause the compiler error "undeclared identifier" at the line testScore=95:

```
testScore = 95;
int testScore;
```

As mentioned earlier in the "Declaring Variables" section, this compiler error will occur even though the variable is declared in the very next line because the compiler reads the code from top to bottom, so when it reaches the line testScore = 95, it has not seen the variable declaration.

The value assigned need not be a literal value, such as 95. The following code assigns to one integer variable the value of another integer variable.

```
int a, b;
a = 44;
b = a;
```

The assignment takes place in two steps:

- First, the value 44 is assigned to the variable *a.*
- Second, the value of *a,* which now is 44, is assigned to the variable *b.*

You also can assign a value to several variables at once. The following code assigns 0 to three integer variables:

```
int a, b, c;
a = b = c = 0;
```

The assignment takes place in three steps, from right to left:

1. The value 0 is assigned to the variable *c*.
2. The value of the variable *c*, which now is 0, is next assigned to the variable *b*.
3. The value of the variable *b*, which now is 0, is assigned to the variable *a*.

Finally, you can assign a value to a variable after it has already been assigned a value. The word "variable" means likely to change or vary. What may change or vary is the variable's value. The following code demonstrates a change in the value of a variable that was previously assigned a value:

```
#include <iostream>
using namespace std;
int main(void)
{
   int testScore;
   testScore = 95;
   cout << "Your test score is " << testScore << "\n";
   testScore = 75;
   cout << "Your test score now is " << testScore << "\n";
   return 0;
}
```

The output is

```
Your test score is 95
Your test score now is 75
```

Assigning a "Compatible" Data Type

The value assigned to a variable must be compatible with the data type of the variable that is the target of the assignment statement. Compatibility means, generally, that if the variable that is the target of the assignment statement has a numeric data type, then the value being assigned must also be a number.

The following code is an example of incompatibility. If it is placed in a program, it will cause a compiler error.

```
int testScore;
testScore = "Jeff";
```

The description of the compiler error is "cannot convert from 'const char [5]' to 'int'." This is the compiler's way of telling you that you are trying to assign a string to an integer, which of course won't work; "Jeff" cannot represent an integer.

The value being assigned need not necessarily be the exact same data type as the variable to which the value is being assigned. In the following code, a floating-point value, 77.83, is being assigned to an integer variable, *testScore*. The resulting output is "The test score is 77."

```
#include <iostream>
using namespace std;
int main(void)
{
   int testScore;
   testScore = 77.83;
   cout << "The test score is " << testScore << "\n";
   return 0;
}
```

While the code runs, data is lost, specifically the value to the right of the decimal point. .83. The fractional part of the number cannot be stored in *testScore,* that variable being a whole number.

Overflow and Underflow

You may recall from Chapter 2 that the short data type has a range from –32768 to 32767. You can run the following program to see what happens when you attempt to assign to a variable a value that is compatible (here a whole number for a short data type) but that is outside its range.

```
#include <iostream>
using namespace std;
int main(void)
{
   short testScore;
   testScore = 32768;
   cout << "Your test score is " << testScore << "\n";
   return 0;
}
```

The output is "Your test score is –32768." That's right, not 32768, but –32768.

This is an example of *overflow.* Overflow occurs when a variable is assigned a value too large for its range. The value assigned, 32768, is 1 too large for the short data type. Therefore, the value overflows and wraps around to the data type's lowest possible value, –32768.

Similarly, an attempt to assign to testScore 32769, which is 2 too large for the short data type, would result in an output of –32767, an attempt to assign to testScore 32770, which is 3 too large for the short data type, would result in an output of –32766, and so on. Figure 3-2 illustrates how the overflow value is reached.

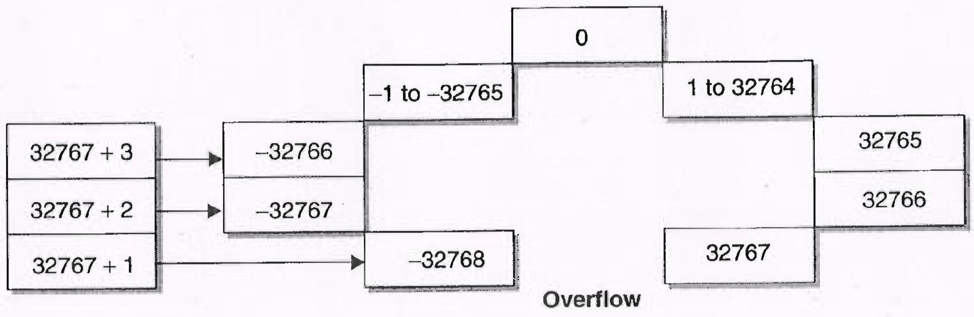


Figure 3-2 Overflow

The converse of overflow is *underflow.* Underflow occurs when a variable is assigned a value too small for its range. The output of the following code is "Your test score is 32767." The value assigned, –32769, is 1 too small for the short data type. Therefore, the value underflows and wraps around to the data type's highest possible value, 32767.

```
#include <iostream>
using namespace std;
int main(void)
{
   short testScore;
   testScore = -32769;
   cout << "Your test score is " << testScore << "\n";
   return 0;
}
```

Similarly, an attempt to assign to testScore –32770, which is 2 too small for the short data type, would result in an output of 32766, an attempt to assign to testScore –32771, which is 3 too small for the short data type, would result in an output of 32765, and so on. Figure 3-3 illustrates how the underflow value is reached.

NOTE: *Floating-point variables, of the float or double data type, also may overflow or underflow. However, the result depends on the compiler used, and may be a run-time error stopping your program, or instead an incorrect result.*

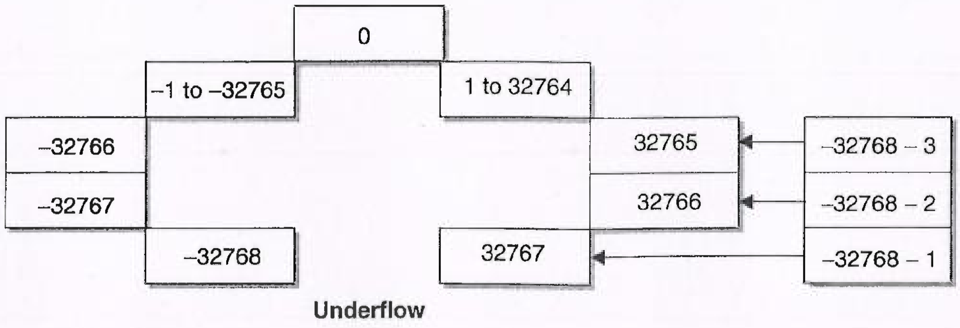


Figure 3-3 Underflow

Using the cin Object

Thus far, the programmer has supplied the values that are assigned to variables. However, most programs are interactive, asking the user to provide information, which the user then inputs, usually via the keyboard.

In Chapter 1, we used the cout object to output information to a standard output, usually the monitor. Now we will use the *cin* object to obtain information from standard input, which usually is the keyboard. The cin object, like the cout object, is defined in the standard library file <iostream>, which therefore must be included (with an include directive) if your code uses cin.

The syntax of a cin statement is

```
cin >> [variable name];
```

The cin object is followed by >>, which is the stream extraction operator. It obtains the input, usually from the keyboard, and assigns that input to the variable to its right.

TIP: *Knowing when to use >> instead of << can be confusing. It may be helpful to remember that the >> and << operators each point in the direction that data is moving. For example in the expression cin >> var, data is moving from standard input into the variable var. By contrast, in the expression cout >> var, the << indicates that data is moving from the variable var to standard output.*

When your program reaches a cin statement, its execution halts until the user types something at the keyboard and presses the ENTER key. Try running the following program. You will see a blinking cursor until you type a number. Once you type a number and press ENTER, the program will output "Your test score is" followed by the number you inputted. For example, if you inputted 100, the output will be "Your test score is 100."

```
#include <iostream>
using namespace std;
int main(void)
{
   int testScore;
   cin >> testScore;
   cout << "Your test score is " << testScore << "\n";
   return 0;
}
```

This program is not very user friendly. Unless the user happened to know what your program did, they would not know what information is being asked of them. Accordingly, a cin statement usually is preceded by a cout statement telling the user what to do. This is called a *prompt.* The following code adds a prompt:

```
#include <iostream>
using namespace std;
int main(void)
{
   int testScore;
   cout << "Enter your test score: ";
   cin >> testScore;
   cout << "Your test score is " << testScore << "\n";
   return 0;
}
```

The program input and output could be

```
Enter your test score: 78
Your test score is 78
```

Assigning a "Compatible" Data Type

As with the assignment operator, the value being assigned by the cin operator need not necessarily be the exact same data type as that of the variable to which the value is being assigned. In the previous program, entering a floating-point value, 77.83, at the prompt for entry of the test score results in the following output: "The test score is 77." Data is lost, though, specifically the part of the number to the right of the decimal point. The cin statement will not read the part of the number to the right of the decimal point because it cannot be stored in a whole number variable.

However, the value being assigned by the cin operator must be compatible with the data type of the variable to which the value is being assigned. In the preceding program, typing "Jeff" at the prompt for entry of the test score results in the following output: "Your test score is –858993460."

Obviously, –858993460 is not a test score anyone would want. Less obvious is the reason why that number is outputted.

The string literal "Jeff" cannot be assigned to an integer variable such as *testScore.* Therefore, the cin operator will not assign "Jeff" to that integer variable. Therefore, when the cout statement attempts to output the value of *testScore,* that variable has not yet been assigned a value.

When *testScore* was declared, there was some value at its memory address left over from programs previously run on the computer. The cout statement, when trying to output the value of *testScore,* does the best it can and attempts to interpret this leftover value. The result of that interpretation is –858993460.

NOTE: ***Compile Time vs. Run-Time Difference When Incompatible Data Types Are Assigned****—Earlier in this chapter, the attempt to assign "Jeff" to testScore (testScore = "Jeff";) resulted in a compiler error. Here, the attempt to assign "Jeff" to testScore using a cin statement instead results in an incorrect value. The reason that this time there is no compiler error is because the value the user would input could not be known at compile time, but instead would be known only at run time. Therefore, there would be no compile error, since at the time of compilation there was no attempt to assign an incompatible value.*

Inputting Values for Multiple Variables

If you are inputting values for several variables, you could input them one line at a time.

```
#include <iostream>
using namespace std;
int main(void)
{
   int myWeight, myHeight;
   string myName;
   cout << "Enter your name: ";
   cin >> myName;
   cout << "Enter your weight in pounds: ";
   cin >> myWeight;
   cout << "Enter your height in inches: ";
   cin >> myHeight;
   cout << "Your name score is " << myName << "\n";
   cout << "Your weight in pounds is " << myWeight << "\n";
   cout << "Your height in inches is " << myHeight << "\n";
   return 0;
}
```

The output of the program, with the input of "Jeff" for the name, 200 for the pounds, and 72 for the height, is

```
Enter your name: Jeff
Enter your weight in pounds: 200
Enter your height in inches: 72
Your name is Jeff
Your weight in pounds is 200
Your height in inches is 72
```

Instead of having separate prompts and cin statements for each variable, you can have one cin statement assign values to all three variables. The syntax is

```
cin >> [first variable] >> [second variable] >>
       [third variable];
```

The same syntax would work when using one cin statement to assign values to four or more variables. The variables are separated by the stream extraction operator >>.

When you use one cin statement to assign values to multiple variables, the user separates each input by one or more spaces. The space tells the cin object that you have finished assigning a value to one variable and the next input should be assigned to the next variable in the cin statement. As before, the user finishes input by choosing the ENTER key.

The following program uses one cin statement to assign values to three variables:

```
#include <iostream>
using namespace std;
#include <string>
int main(void)
{
   int myWeight, myHeight;
   string name;
   cout << "Enter your name, weight in pounds and height
           in inches\n";
   cout << "The three inputs should be separated by a
           space\n";
   cin >> name >> myWeight >> myHeight;
   cout << "Your name is " << name << "\n";
   cout << "Your weight in pounds is " << myWeight << "\n";
   cout << "Your height in inches is " << myHeight << "\n";
   return 0;
}
```

The interaction between user input and the cin statement could be as follows:

- The user would type "Jeff," followed by a space.

- The space tells the cin object that the first input has ended, so the cin object will assign “Jeff” to the first variable in the cin statement, *name*.
- The user would type 200, followed by a space.
- The space tells the cin object the second input has ended, so the cin object will assign 200 to the next variable in the cin statement, *myWeight*.
- The user would type 200, and then press the ENTER key.
- The ENTER key tells the cin object that the third and final input has ended, so the cin object will assign 72 to the remaining variable in the cin statement, *myHeight,* which completes execution of the cin statement.

The resulting program output would be

```
Enter your name, weight in pounds and height in inches
The three inputs should be separated by a space
Jeff 200 72
Your name is Jeff
Your weight in pounds is 200
Your height in inches is 72
```

Assigning a “Compatible” Data Type

The data types in the cin statement may be different. In this example, the data type of the first variable is a string, whereas the data type of the second and third variables is an integer.

What is important is that the order of the input matches the order of the data types of the variables in the cin statement. The input order “Jeff,” 200, and 72 is assigned to the variables in the order of their appearance in the cin statement, *myName, myWeight,* and *myHeight.* Therefore, “Jeff” is assigned to the string variable *myName,* 72 to the integer variable *myWeight,* and 200 to the integer variable *myHeight.*

The importance of the order of the input matching the order of the data types of the variables in the cin statement is demonstrated by changing the order of the user’s input from “Jeff,” 200, and 72, to 200, “Jeff,” and 72. The program output then would be

```
Enter your name, weight in pounds and height in inches
The three inputs should be separated by a space
200 Jeff 72
Your name is 200
Your weight in pounds is -858993460
Your height in inches is -858993460
```

While I would like to lose weight, –858993460 seems a bit extreme. Also, while it is understandable why "Jeff" cannot be assigned to my weight, 72 was not assigned to my height either.

The one output that is correct is the name. Any characters, including digits, can be part of a string. Therefore, while 200 may be an unusual name to us, it is perfectly OK for cin, which therefore assigns 200 to the string variable *name.*

Why –858993460 was outputted for myWeight also has been explained earlier in the example in which the user entered "Jeff" at the prompt to enter a test score.

However, 72 would be a valid value for assignment to the integer variable *myHeight.* Why then isn't 72 the output for height?

The reason is that the next value for cin to assign is not 72, but instead "Jeff." Since cin was unable to assign "Jeff" to *myWeight,* the value "Jeff" remains next in line for assignment, this time to the variable *myHeight.* Unfortunately, cin is unable to assign "Jeff" to *myHeight* either, so the value of *myHeight,* like *myWeight,* also is outputted as –858993460.

Inputting Multiple Words into a String

Finally, cin will only take the first word of a string. If in the following program you input "Jeff Kent" at the prompt, the output will be "Your name is Jeff" not "Your name is Jeff Kent."

```
#include <iostream>
using namespace std;
#include <string>
int main(void)
{
   string name;
   cout << "Enter your name: ";
   cin >> name;
   cout << "Your name is " << name;
   return 0;
}
```

The reason why the value of name is outputted only as "Jeff," omitting "Kent," is that the cin object interprets the space between "Jeff" and "Kent" as indicating that the user has finished inputting the value of the name variable.

The solution involves using either the get or getline method of the cin object. These methods will be covered in Chapter 10.

Overflow and Underflow

The consequences of an overflow or underflow of whole number variables is more unpredictable with cin than with the assignment operator. Inputting either 32768, which is 1 more than the highest number in the range of a short data type, or –32769, 1 less than the lowest number in that range, results on my computer in the output "Your test score is –13108."

```
#include <iostream>
using namespace std;
int main(void)
{
   short testScore;
   testScore = 32768;
   cout << "Your test score is " << testScore << "\n";
   return 0;
}
```

Summary

A variable serves two purposes. It provides you with a way of referring to particular information, and also reserves the amount of memory necessary to store that information.

You must create a variable before you can start using it. You create a variable by declaring it. You may declare multiple variables of the same type in one statement.

You can use the address operator, &, to determine the address of a variable, and the sizeof operator to determine the size of a variable.

The purpose of a variable is to store information. Therefore, after you have created a variable, the next logical step is to specify the information that the variable will store. This is called *assigning* a value to a variable.

A variable can be assigned a value either by the programmer in code or by the user, usually via the keyboard, when the program is running. You use the assignment operator to assign a value supplied by code. You use the cin object to assign a value supplied by the user.

In the next chapter, you will learn how to use variables to perform arithmetic.

Quiz

1. What is the effect of declaring a variable?
2. Can you refer to a variable before declaring it as long as you declare it later?
3. Can you declare several variables in the same statement?
4. What is a "naming convention" with respect to variables?
5. What is the difference between the address and sizeof operators?
6. What is initialization?
7. What is overflow?
8. What is the consequence of using an assignment operator to assign a string value to an integer variable?
9. Do you use the cin object for compile time or run-time assignment of values to variables?
10. Can you use one cin statement to assign values to several variables of different data types?

CHAPTER 5

Making Decisions: if and switch Statements

The famous poem "The Road Not Taken" by Robert Frost begins: "Two roads diverged in a yellow wood, and sorry I could not travel both." This poem illustrates that life, if nothing else, presents us with choices.

Similarly, computer programs present their users with choices. So far, for the sake of simplicity, the flow of each program has followed a relatively straight line, taking a predetermined path from beginning to end. However, as programs become more sophisticated, they often branch in two or more directions based on a choice a user makes. For example, when I am buying books online, I am presented with choices such as adding another item to my shopping cart, recalculating my total, or checking out. The program does something different if I add another item to my shopping cart rather than check out.

The program determines the action it takes by comparing my choice with the various alternatives. That comparison is made using a relational operator. There are relational operators to test for equality, inequality, whether one value is greater (or less) than another, and other comparisons.

The code then needs to be structured so different code executes depending on which choice was made. This is done using either the if statement or the switch case statement, both of which we'll discuss in this chapter.

We'll also discuss flowcharting, which enables you to visually depict the flow of a program. Flowcharting becomes increasingly helpful as we transition from relatively simple programs that flow in a straight line to more complex programs that branch in different directions.

Relational Operators

We make comparisons all the time, and so do programs. A program may need to determine whether one value is equal to, greater than, or less than another value. For example, if a program calculates the cost of a ticket to a movie in which children less than 12 get in free, it needs to find out if the customer's age is less than 12.

Programs compare values by using a relational operator. Table 5-1 lists the relational operators supported by C++:

Operator	**Meaning**
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

Table 5-1 Relational Operators

Relational Expressions

Like the arithmetic operators discussed in the last chapter, the relational operators are binary—that is, they compare two operands. A statement with two operands and a relational operator between them is called a *relational expression.*

The result of a relational expression is a Boolean value, depicted as either true or false. Table 5-2 lists several relational expressions, using different relational operators and their values.

Relational Expression	Value
4 == 4	true
4 < 4	false
4 <= 4	true
4 > 4	false
4 != 4	false
4 == 5	false
4 < 5	true
4 <= 5	true
4 >= 5	false
4 != 5	true

Table 5-2 Relational Expressions and Their Values

Table 5-2 uses operands that have literal values. A literal value is a value that cannot change. 4 is a literal value, and cannot have a value other than the number 4.

Operands may also be variables (which were discussed in Chapter 3). The following program outputs the results of several variable comparisons.

```
#include <iostream>
using namespace std;
int main(void)
{
   int a = 4, b = 5;
   cout << a << " > " << b << " is " << (a > b) << endl;
   cout << a << " >= " << b << " is " << (a >= b) << endl;
   cout << a << " == " << b << " is " << (a == b) << endl;
```

```
    cout << a << " <= " <<  b << " is " << (a <= b) << endl;
    cout << a << " < " <<  b << " is " << (a < b) << endl;
    return 0;
}
```

The program's output is

```
4 > 5 is 0
4 >= 5 is 0
4 == 5 is 0
4 <= 5 is 1
4 < 5 is 1
```

In the output, 0 is false and 1 is true. 0 is the integer value of Boolean false, while 1 is the usual integer value of Boolean true. As you may recall from Chapter 1, early computers consisted of wires and switches in which the electrical current followed a path that depended on which switches were in the on position (corresponding to the value one) or the off position (corresponding to the value zero). The on position corresponds to Boolean true, the off position to Boolean false.

CAUTION: *While the usual integer value of logical true is 1, any non-zero number may be logical true. Therefore, in a Boolean comparison, do not compare a value to 1, compare it to true.*

The data types of the two operands need not be the same. For example, you could change the data type of the variable *b* in the preceding program from an int to a float and the program still would compile and provide the same output. However, the data types of the two operands need to be compatible. As you may recall from Chapter 3, compatibility means, generally, that if one of the variable operands in the relational expression is a numeric data type, then the expression's other variable operand must also be a numeric data type.

For example, the program would not compile if you changed the data type of the variable *b* in the preceding program from an int to a string.

Precedence

Relational operators have higher precedence than assignment operators and lower precedence than arithmetic operators. Table 5-3 lists precedence among relational operators.

Precedence	Operator
Highest	> >= < <=
Lowest	== !=

Table 5-3 Precedence of Relational Operators

Operators in the same row have equal precedence. The associativity of relational operators of equal precedence is from left to right.

Flowcharting

A program, like a river, flows from beginning to end. Programmers may find it helpful, both in writing code and in understanding someone else's code, to visually depict the flow of the program. After all, as the adage goes, a picture is worth a thousand words. The ability to visualize the flow of a program becomes even more helpful as we transition from relatively simple programs that flow in a straight line to more complex varieties that branch in different directions based on the value of a relational expression.

Programmers use a flowchart to visually depict the flow of a program. Flowcharts use standardized symbols prescribed by the American National Standard Institute (ANSI), which prescribes other standards we will be using in this book. These flowcharting symbols represent different aspects of a program, such as the start or end of a program, user input, how it displays on a monitor, and so on. These symbols are joined by arrows and other connectors which show the connections between different parts of the program and the direction of the program flow. Figure 5-1 shows several commonly used flowchart symbols. Others will be introduced later in this book as they are used.

The following program from Chapter 4 can be depicted with a flowchart. As you may recall, this program first assigns to the integer variable *total* the value inputted by the user for the number of preregistered students. The program then assigns to the integer variable *added* the value inputted by the user for the number of students adding the course. The program then uses the addition operator to add two operands, *total* and *added.* The resulting sum is then assigned to *total,* which now reflects the

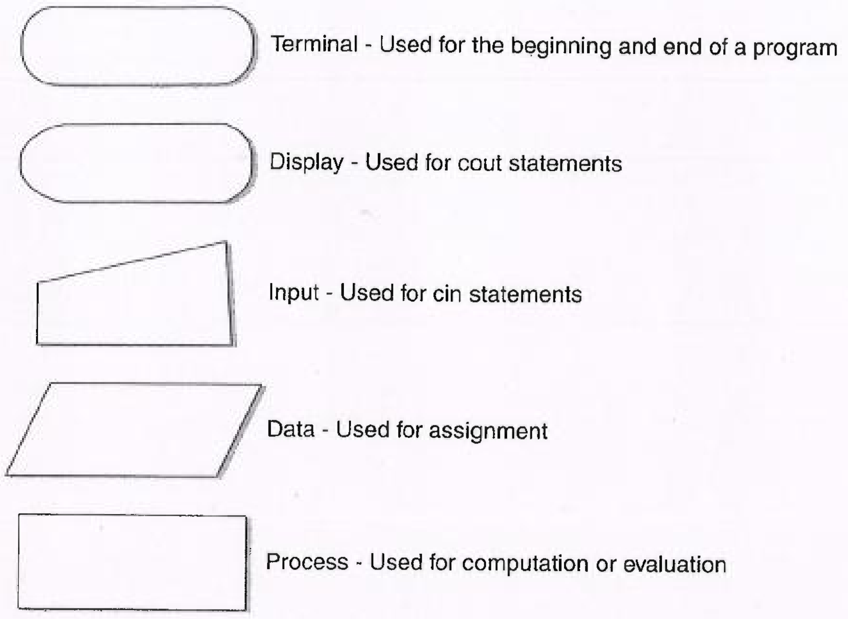

Terminal - Used for the beginning and end of a program
Display - Used for cout statements
Input - Used for cin statements
Data - Used for assignment
Process - Used for computation or evaluation

Figure 5-1 Commonly used flowchart symbols

total number of students in the course, both preregistered and added. That sum then is outputted.

```
#include <iostream>
using namespace std;
int main(void)
{
   int total, added;
   cout << "Enter number of pre-registered students: ";
   cin >> total;
   cout << "Enter number of students adding the course: ";
   cin >> added;
   total = total + added;
   cout << "Total number of students: " << total;
   return 0;
}
```

Figure 5-2 shows a flowchart of this program.

This program was relatively linear. By contrast, the following programs will branch in different directions based on the value the user inputs. We will use flowcharts in later sections of this chapter to help explain how different code executes depending on the result of comparisons with the user's input.

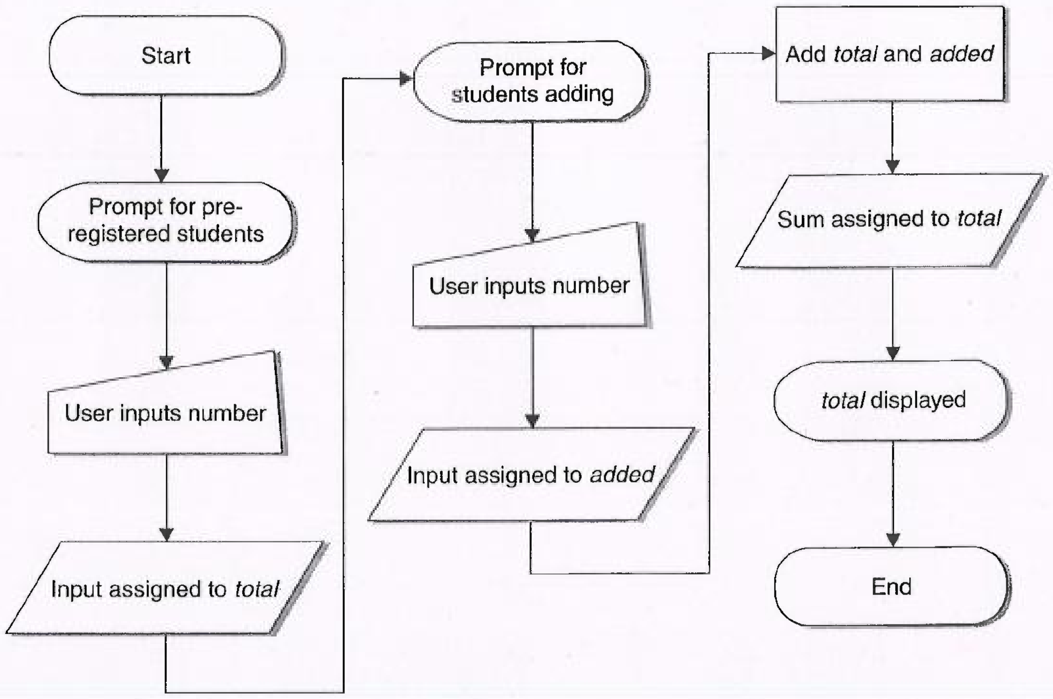

Start
Prompt for pre-registered students
User inputs number
Input assigned to *total*
Prompt for students adding
User inputs number
Input assigned to *added*
Add *total* and *added*
Sum assigned to *total*
total displayed
End

Figure 5-2 Flowchart of the program adding preregistered and added students

The if Statement

The if statement is used to execute code only when the value of a relational expression is true. The syntax of an if statement is

```
if (Boolean value)
   statement;
```

Both lines together are called an if statement. The first line consists of the if keyword followed by an expression, such as a relational expression, that evaluates to a Boolean value, true or false. The relational (or other Boolean) expression must be in parentheses, and should not be terminated with a semicolon.

The next line is called a conditional statement. As you may recall from Chapter 1, a statement is an instruction to the computer, directing it to perform a specific action. The statement is conditional because it executes only if the value of the relational expression is true. If the value of the relational expression is false, then the conditional statement is not executed—meaning, it's essentially skipped.

The following program, which tests if a whole number entered by the user is even, illustrates the use of an if statement.

```
#include <iostream>
using namespace std;
int main(void)
{
   int num;
   cout << "Enter a whole number: ";
   cin >> num;
   if ( num % 2 == 0 )
      cout << "The number is even" << endl;
   return 0;
}
```

If the user enters an even number, then the program outputs that the number is even.

```
Enter a whole number: 16
The number is even
```

However, if the user enters an odd number, then there is no output that the number is even.

```
Enter a whole number: 17
```

Figure 5-3 is a flowchart of this program. This flowchart has one new symbol: a diamond. It's used to represent the true/false statement being tested.

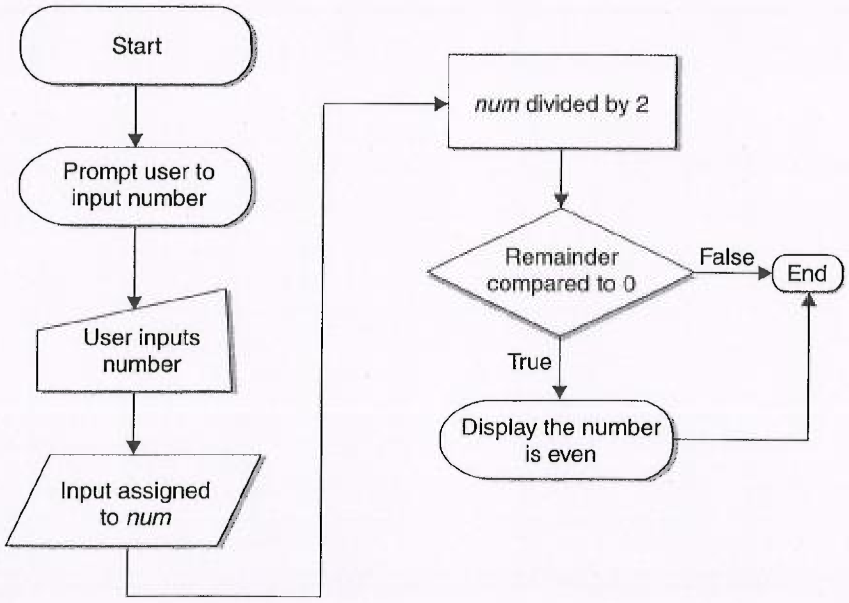


Figure 5-3 Flowchart of a program that determines whether a number is even

Let's now analyze how the program works. You may find the flowchart a helpful visual aid in following this textual explanation.

The program first prompts the user to enter a number. It then stores that input in the integer variable *num.*

The program next evaluates the relational expression *num % 2 == 0,* which is enclosed in parentheses following the *if* keyword. That expression involves two operators, the arithmetic modulus operator (%) and the relational equality operator (===). Since arithmetic operators have higher precedence than relational operators, the expression *num % 2* will be evaluated first, with the result then compared to zero.

A number is even if, when divided by two, the remainder equals zero. You learned in Chapter 4 that the modulus operator will return the remainder from integer division. Accordingly, the expression *num % 2* will divide the number entered by the user by two, and return the remainder. That remainder then will be compared to zero using the relational equality operator.

If the relational expression is true, which it would be if the number inputted by the user is even, then the conditional statement executes, outputting "The number is even." If the relational expression is false, which it would be if the number inputted by the user is odd, then the conditional statement is skipped, and it will not execute.

Indenting

It is good practice to indent the conditional statement.

```
if ( num % 2 == 0 );   // don't put a semicolon here!
   cout << "The number is even" << endl;
```

While the compiler doesn't care whether you indent or not, indentation makes it easier for you, the programmer, to see that the statement is conditional.

Common Mistakes

During several years of teaching C++ in an introductory programming class, I have noticed several common mistakes in the writing of if statements. Some of these mistakes may result in compiler errors and therefore are easy to spot. However, other mistakes are harder to pick out since they do not cause an error, either at compile time or run-time, but instead give rise to illogical results.

Don't Put a Semicolon after the Relational Expression!

The first common mistake is to place a semicolon after the relational expression:

```
if ( num % 2 == 0 );  // don't put a semicolon here!
   cout << "The number is even" << endl;
```

Since the compiler generally ignores blank spaces, the following if statement would be the same, and better illustrates visually the problem:

```
if ( num % 2 == 0 )
   ;  // don't put a semicolon here!
cout << "The number is even" << endl;
```

No compiler error will result. The compiler will assume from the semicolon that it is an empty statement. An empty statement does nothing, and though it is perfectly legal in C++, and indeed sometimes has a purpose, here it is not intended.

One consequence will be that the empty statement will execute if the relational expression is true. If this comes about, nothing will happen. So far, there is no harm done.

However, there is an additional consequence, an illogical result. The cout statement "The number is even" will execute whether or not the relational expression is true. In other words, even if an odd number is entered, the program will output "The number is even."

```
Enter a whole number: 17
The number is even
```

The reason the cout statement will execute whether or not the relational expression is true is that the cout statement no longer is part of the if statement. Unless you use curly braces as explained in the next section, only the first statement following the if keyword and relational expression is conditional. That first conditional statement is the empty statement, by virtue of the semicolon following the if expression.

Curly Braces Needed for Multiple Conditional Statements

As just discussed, unless you use curly braces (explained later in this section), only the first statement following the if keyword and relational expression is conditional. For example, in the following code, only the first cout statement is conditional. The second cout statement is not, so it will execute whether the relational expression is true or false:

```
if ( num % 2 == 0 )
   cout << "The number is even" << endl;
cout << "And the number is not odd" << endl;
```

NOTE: The indentation tells the programmer which statement is conditional and which is not. The compiler ignores indentation.

Thus, if the user enters an odd number such as 17, the cout statement "The number is even" will not display because the relational expression is false. However, the following statement "And the number is not odd" will display because that statement does not belong to the if statement.

```
Enter a whole number: 17
And the number is not odd
```

If you want more than one statement to be part of the overall if statement, you must encase these statements in curly braces:

```
   if ( num % 2 == 0 )
   {
      cout << "The number is even" << endl;
      cout << "And the number is not odd" << endl;
   }
```

Now the second cout statement will execute only if the if expression is true.

Forgetting these curly braces when you want multiple statements to be conditional is another common syntax error.

Don't Mistakenly Use the Assignment Operator!

The third most common syntax error is to use the assignment operator instead of the relational equality operator because the assignment operator looks like an equal sign:

```
   if ( num % 2 = 0 )   // wrong operator!
      cout << "The number is even" << endl;
```

The result is that the if expression will not evaluate as the result of a comparison. Instead, it will evaluate the expression within the parentheses as the end result of the assignment, with a non-zero value being regarded as true, a zero value being regarded as false.

NOTE: Some compilers will treat this mistake as a compiler error.

The if / else Statement

One problem with the program that tests whether a number is even is that there is no output if the number is odd. While there is a conditional statement if the relational expression is true, there is no corresponding conditional statement (cout << "The number is odd") if the relational expression is false.

The solution is to add an else part to the if statement. The result is an if / else statement. The syntax of an if / else statement is

```
if (relational expression)
   conditional statement;
else
   conditional statement;
```

Accordingly, the program may be modified to add an else part to the if statement:

```
#include <iostream>
using namespace std;
int main(void)
{
   int num;
   cout << "Enter a whole number: ";
   cin >> num;
   if ( num % 2 == 0 )
      cout << "The number is even" << endl;
   else
      cout << "The number is odd" << endl;
   return 0;
}
```

Run this code. If the inputted number is even, then the output once again is "The number is even." However, if the number is now odd, instead of no output, the output is "The number is odd."

```
Enter a whole number: 17
The number is odd
```

Figure 5-4 uses a flowchart to illustrate this program.

Conditional Operator

This program could be rewritten using the conditional operator.

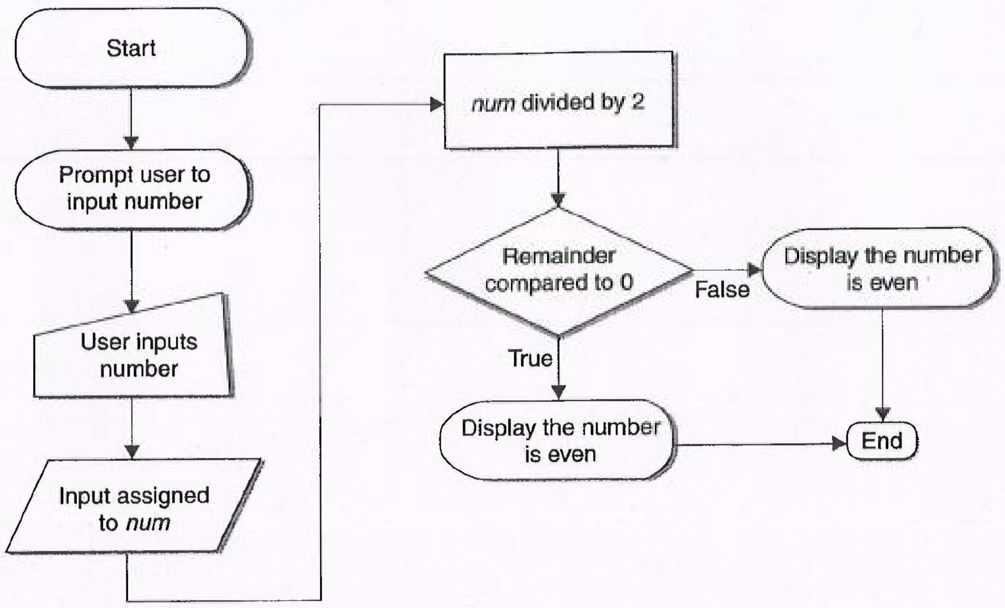


Figure 5-4 Flowchart of program output if number is even or odd

```
#include <iostream>
using namespace std;
int main(void)
{
   int num;
   cout << "Enter a whole number: ";
   cin >> num;
   cout << "The number is " << ( num % 2 == 0 ? "even" :
            "odd") << endl;
   return 0;
}
```

The syntax of the conditional operator is

```
[Relational expression] ? [statement if true] :
[statement if false]
```

In this example, the relational expression is *num % 2 == 0*. If the value of the relational expression is true, then the output is "even." However, if the value of the relational expression is false, then the output is "odd."

The conditional operator requires three operands, the relational expression and the two conditional statements. Therefore, it is considered a ternary operator.

Common Mistakes

Just as with the if statement, I noticed several common syntax mistakes with the else statement while teaching C++ in introductory programming classes.

No else Without an if

You can have an if expression without an else part. However, you cannot have an else part without an if part. The else part must be part of an overall if statement. This requirement is logical. The else part works as "none of the above"; without an if part there is no "above."

As a consequence, placing a semicolon after the Boolean expression following the if keyword will result in a compiler error. Since curly braces are not used, the if statement ends after the empty statement created by the incorrectly placed semicolon. The cout statement "The number is even" is not part of the if statement. Consequently, the else part is not part of the if statement, and therefore will be regarded as an else part without an if part.

```
if ( num % 2 == 0 ); // don't put a semicolon here
   cout << "The number is even" << endl;
else ( num % 2 == 1 )
   cout << "The number is odd" << endl;
```

Don't Put a Relational Expression after the else Keyword!

Another common mistake is to place a relational expression in parentheses after the else keyword. This will not cause a compiler or run-time error, but it will often cause an illogical result.

```
if ( num % 2 == 0 )
   cout << "The number is even" << endl;
else ( num % 2 == 1 )
   cout << "The number is odd" << endl;
```

The program will not compile, and the cout statement following the else expression will be highlighted with an error description such as "missing ';' before identifier 'cout'."

Actually, the error description is misleading. There is nothing wrong with the cout statement. Instead, no relational expression should follow the else keyword. The reason is that the else acts like "none of the above" in a multiple choice test.

If the if expression is not true, then the conditional statements connected to the else part execute.

Don't Put a Semicolon after the Else!

Another common mistake is to place a semicolon after the else expression. This too will not cause a compiler or run-time error, but often will cause an illogical result.

For example, in the following code, the cout statement "The number is odd" will output even if the number that's input is even.

```
if ( num % 2 == 0 )
   cout << "The number is even" << endl;
else;  // don't put a semicolon here!
   cout << "The number is odd" << endl;
```

The result of inputting an even number will be

```
Enter a whole number: 16
The number is even
The number is odd
```

The cout statement "The number is odd" will execute whether or not the relational expression is true because the cout statement no longer is part of the if statement. Unless you use curly braces as explained already in connection with the if statement, only the first statement following the else keyword is conditional. That first, conditional statement is the empty statement by virtue of the semicolon following the if expression. Therefore, the cout statement "The number is odd" is not part of the if statement at all.

Curly Braces Are Needed for Multiple Conditional Statements

As with the if expression, if you want more than one conditional statement to belong to the else part, then you must encase the statements in curly braces. For example, in the following code fragment, the cout statement "This also belongs to the else part" will always display whether the number is even or odd since it does not belong to the if statement.

```
if ( num % 2 == 0 )
   cout << "The number is even" << endl;
else
   cout << "The number is odd" << endl;
cout << "This also belongs to the else part";
```

The sample input and output could be

```
Enter a whole number: 16
The number is even
This also belongs to the else part
```

Encasing the multiple conditional statements in curly braces solves this issue.

```
if ( num % 2 == 0 )
      cout << "The number is even" << endl;
else
{
      cout << "The number is odd" << endl;
      cout << "This also belongs to the else part";
}
```

The if /else if /else Statement

The program we used to illustrate the if/else statement involved only two alternatives. Additionally, these alternatives were mutually exclusive; only one could be chosen, not both. A whole number is either even or odd; it can't be both and there is no third alterative. There are many other examples of only two mutually exclusive alternatives. For example, a person is either dead or alive, male or female, child or adult.

However, there are other scenarios where there are more than two, mutually exclusive alternatives. For example, if you take a test, your grade may be one of five types: A, B, C, D, or F. Additionally, these grades are mutually exclusive; you can't get an A and a C on the same test.

Since you can have only one if expression and only one else expression in an if statement, you need another expression for the third and additional alternatives. That expression is else if.

You use the if / else if / else statement when there are three or more mutually exclusive alternatives. The if / else if / else statement has an if part and an else part, like an if/else statement. However, it also has one or more else if parts.

NOTE: While the if part is required, the else part is not. Without it, the statement would be named an if / else if statement.

The else if part works similarly to an if expression. The else if keywords are followed by a relational expression. If the expression is true, then the conditional statement or statements "belonging" to the else if part execute. Otherwise, they don't.

While an if statement may include only one if part and one else part, it may include multiple else if parts.

The following program shows the if /else if /else statement in action in a program that determines your grade based on your test score.

```
#include <iostream>
using namespace std;
int main(void)
{
   int testScore;
   cout << "Enter your test score: ";
   cin >> testScore;
   if (testScore >= 90 )
      cout << "Your grade is an A" << endl;
   else if (testScore >= 80 )
      cout << "Your grade is a B" << endl;
   else if (testScore >= 70 )
      cout << "Your grade is a C" << endl;
   else if (testScore >= 60 )
      cout << "Your grade is a D" << endl;
   else
      cout << "Your grade is an F" << endl;
   return 0;
}
```

Here are several sample runs, each separated by a dotted line:

```
Enter your test score: 77
Your grade is a C
------
Enter your test score: 91
Your grade is an A
------
Enter your test score: 55
Your grade is an F
```

Figure 5-5 uses a flowchart to illustrate this program.

In this program, if your test score is 90 or better, then the conditional statement belonging to the if part executes, displaying that you received an A. The relational expressions of each of the following else if parts also are true; if your score is 90 or better, it also is 80 or better, 70 or better, and so on. However, in an if / else if / else statement, only the conditional statements in the first part whose relational expression is true will execute; the remaining parts are skipped.

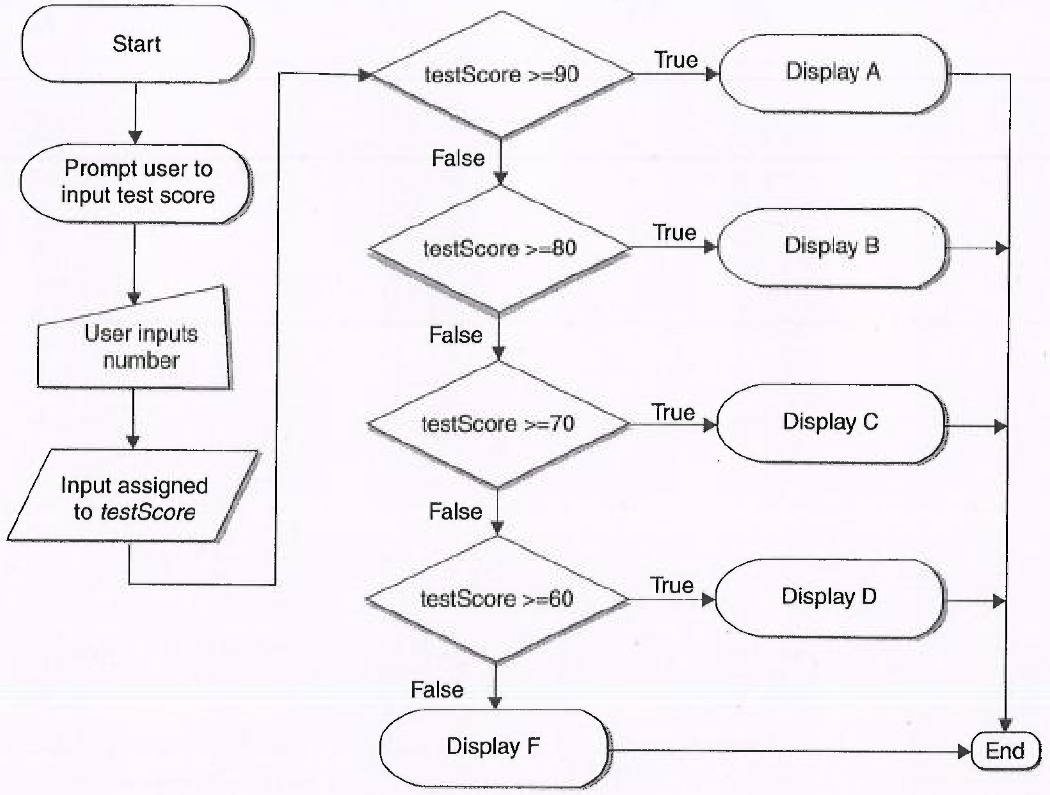


Figure 5-5 Flowchart depiction of grading program

Common Syntax Errors

The common syntax errors for the if part discussed earlier in this chapter apply to the else if part also. Don't put a semicolon after the relational expression, and multiple conditional statements must be enclosed in curly braces.

Additionally, just as you cannot have an else part without a preceding if part, you cannot have an else if part without a preceding if part. However, you may have an if part and one or more else if parts without an else part. The downside in omitting the else part is you will not have code to cover the "none of the above" scenario in which none of the relational expressions belonging to the if part and else if parts is true.

The switch Statement

The switch statement is similar to an if /else if /else statement. It evaluates the value of an integer expression and then compares that value to two or more other values to determine which code to execute.

The following program shows a switch statement in action in a program that determines your average based on your grade:

```
#include <iostream>
using namespace std;
int main(void)
{
   char grade;
   cout << "Enter your grade: ";
   cin >> grade;
   switch (grade)
   {
   case 'A':
      cout << "Your average must be between 90 - 100"
           << endl;
      break;
   case 'B':
      cout << "Your average must be between 80 - 89"
           << endl;
      break;
   case 'C':
      cout << "Your average must be between 70 - 79"
           << endl;
      break;
   case 'D':
      cout << "Your average must be between 60 - 69"
           << endl;
      break;
   default:
      cout << "Your average must be below 60" << endl;
   }
  return 0;
}
```

Here are several sample runs, each separated by a dotted line:

```
Enter your grade: C
Your average must be between 70 - 79
------
Enter your grade: A
Your average must be between 90 - 100
------
Enter your grade: F
Your average must be below 60
```

Figure 5-6 uses a flowchart to illustrate this program.

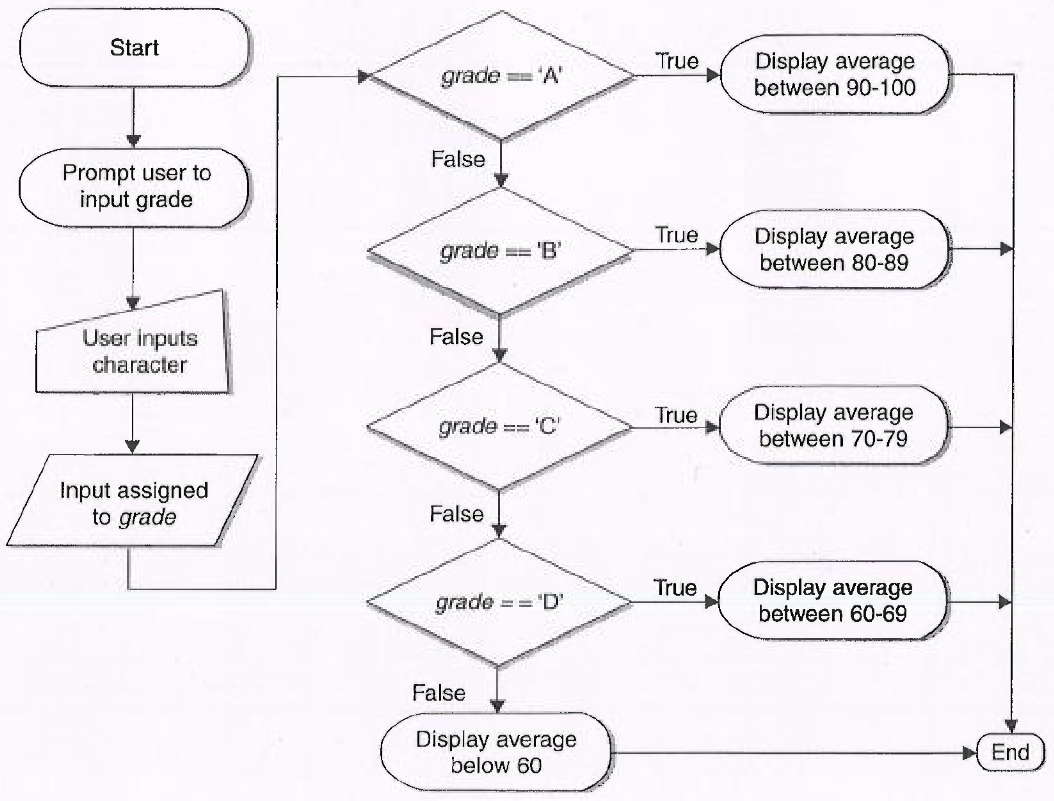

Start
Prompt user to input grade
User inputs character
Input assigned to *grade*
grade == 'A'
True
Display average between 90-100
False
grade == 'B'
True
Display average between 80-89
False
grade == 'C'
True
Display average between 70-79
False
grade = = 'D'
True
Display average between 60-69
False
Display average below 60
End

Figure 5-6 Flowchart depiction of the grade determination program

Let's now analyze the program.

The *switch* keyword evaluates an integer expression, *grade*. While grade is a character variable, every character has a corresponding integer value.

Earlier in this chapter, we discussed flowchart symbols prescribed by the American National Standard Institute (ANSI), and mentioned that ANSI also prescribes other standards that we will be using in this book. One of those other standards is the ANSI character set, which includes 256 characters, each having an integer value between 0 and 255. These values also are called ASCII values, since values 0 to 127 of the ANSI character set are the same as in the ASCII (American Standard Code for Information Interchange) character set.

Table 5-4 lists the ANSI/ASCII values for commonly used characters. Note that digits also can be characters, and that the ANSI/ASCII value of an uppercase character is different than the value of the corresponding lowercase character.

Character	Value
0	48
9	57
A	65
Z	90
a	97
z	122

Table 5-4 Selected ANSI/ASCII Values

Each *case* keyword is followed by an integer expression that must be constant, that is, it cannot change in value during the life of the program. Therefore, a variable cannot follow a case keyword. In this program, the constant is a character literal, such as A, B, and so on. Each character's ANSI value is an integer value, and the integer expression is followed by a colon.

CAUTION: A common mistake is to follow the integer expression not with a colon but with a semicolon, which is typically used to terminate statements. This will cause a compiler error.

The default keyword serves the same purpose as an else part in an if /else if /else statement, and therefore is not followed by an integer expression.

The integer expression following the switch keyword is evaluated and compared with the integer constant following each case keyword, from top to bottom. If there is a *match*—that is, the two integers are equal—then the statements belonging to that case are executed. Otherwise, they are not. Thus, the statements belonging to a case are conditional, just as are statements in an if, else if, or else part. However, unlike an if /else if /else statement, multiple conditional statements belonging to a case do not need to be enclosed in curly braces.

Differences Between switch and if /else if /else Statements

While a switch statement is similar to an if /else if /else statement, there are important differences.

One difference is that in an if/else if/else statement, the comparison following the if part may be independent of the comparison following an else if part. The following example, while perhaps a bit silly, is illustrative of this concept:

```
if (apples == oranges)
   do this;
else if (sales >= 5000)
   do that;
```

By contrast, in a switch statement, the constant integer expression following a case keyword must be compared with the value following the switch keyword, and nothing else. The next chapter on logical operators discusses other differences between switch and if/else if/else statements. However, two differences can be discussed now. One is commonly known as "falling through." The other concerns ranges of numbers.

Falling Through

In an if/else if/else statement, each part is separate from all the others. By contrast, in a switch statement (once a matching case statement is found), unless a break statement is reached, execution "falls through" to the following case statements that execute their conditional statements without checking for a match. For example, if you removed the break statements from the program, you could have the following sample run:

```
Enter your grade: A
Your average must be between 90 - 100
Your average must be between 80 - 89
Your average must be between 70 - 79
Your average must be between 60 - 69
Your average must be below 60
```

This "falling through" behavior is not necessarily bad. In the following modification of the grade program, the falling-through behavior permits the user to enter a lowercase grade in addition to an uppercase grade.

```
#include <iostream>
using namespace std;
int main(void)
{
   char grade;
   cout << "Enter your grade: ";
   cin >> grade;
   switch (grade)
```

```
    {
    case 'a':
    case 'A':
       cout << "Your average must be between 90 - 100"
            << endl;
       break;
    case 'b':
    case 'B':
       cout << "Your average must be between 80 - 89"
            << endl;
       break;
    case 'c':
    case 'C':
       cout << "Your average must be between 70 - 79"
            << endl;
       break;
    case 'd':
    case 'D':
       cout << "Your average must be between 60 - 69"
            << endl;
       break;
    default:
       cout << "Your average must be below 60" << endl;
    }
return 0;
}
```

Another example occurs in the following program. Since the “D” (for deluxe) option includes the feature in the “L” (for leather) option, case ‘D’ deliberately falls through the case ‘L.’

```
#include <iostream>
using namespace std;
int main(void)
{
    char choice;
    cout << "Choose your car\n";
    cout << "S for Standard\n";
    cout << "L for Leather Seats\n";
    cout << "D for Leather Seats + Chrome Wheels\n";
    cin >> choice;
    cout << "Extra features purchased\n";
    switch (choice)
    {
```

```
    case 'D':
       cout << "Chrome wheels\n";
    case 'L':
       cout << "Leather seats\n";
       break;
    default:
       cout << "None selected\n";}
    return 0;
}
```

The sample run could be

```
Choose your car
S for Standard
L for Leather Seats
D for Leather Seats + Chrome Wheels
D
Extra features purchased
Chrome wheels
Leather seats
```

Ranges of Numbers

Another difference between switch and if/else ifelse statements concerns the handling of ranges of numbers. For example, earlier in this chapter we used an if /else if /else statement to output the user's grade based on the test score that was input by the user. The issued grade was an A if the test score was between 90 and 100, a B if the test score was between 80 and 89, and so on. The if /else if /else statement in that program was

```
if (testScore >= 90 )
   cout << "Your grade is an A" << endl;
else if (testScore >= 80 )
   cout << "Your grade is a B" << endl;
else if (testScore >= 70 )
   cout << "Your grade is a C" << endl;
else if (testScore >= 60 )
   cout << "Your grade is a D" << endl;
else
   cout << "Your grade is an F" << endl;
```

By contrast, a case statement cannot be followed by an expression such as testScore >= 90 because the case statement keyword has to be followed by an integer constant. Instead, a case statement would be necessary for each possible test score. The following code fragment shows only the code for an A or B grade to avoid the code example being unduly long, but the code for a C or D grade would be essentially a repeat (an F grade would be handled with the default keyword).

```
switch (testScore)
{
case 100:
case 99:
case 98:
case 97:
case 96:
case 95:
case 94:
case 93:
case 92:
case 91:
case 90:
   cout << "Your grade is an A";
   break;
case 89:
case 88:
case 87:
case 86:
case 85:
case 84:
case 83:
case 82:
case 81:
case 80:
   cout << "Your grade is an A";
   break;
}
```

This code example illustrates that the switch statement is more cumbersome than the if /else if /else structure in dealing with ranges of numbers.

Summary

Computer programs usually do not take a preordained path from beginning to end. Instead, different code executes based on choices made by the user. Relational operators are used to compare the user's choice with various alternatives. The if, if/else, if /else if /else, and switch statements are used to structure the code so different code executes depending on which choice was made. You also learned about flowcharts, which help make programs more understandable by visually depicting the program components and flow.

In this chapter, only one comparison was made at a time. However, sometimes more than one comparison needs to be made. For example, you are eligible to vote in the U.S. only if you are a citizen and are at least 18 years old. You cannot vote unless both are true. However, you may get into a movie free if you are either a senior citizen (65 years or older) or a child (12 or under). Thus, you get in free if either is true. In the next chapter, you will learn about how to use logical operators to combine comparisons.

Quiz

1. How many operands are in a relational expression?
2. What is the purpose of a flowchart?
3. What is the data type of the expression following the if keyword?
4. In an if /else if /else statement, which part must you have one, but only one, of?
5. In an if /else if /else statement, which part may you have more than one of?
6. In an if /else if /else statement, which part may you omit?
7. In a switch statement, what is the required data type of expression following the switch keyword?
8. In a switch statement, may an expression of the character data type follow the switch keyword?
9. In a switch statement, may the expression following a case keyword be a variable?
10. Which keyword in a switch statement corresponds to the else keyword in an if /else if /else statement?

CHAPTER 6

Nested if Statements and Logical Operators

Chapter 5 began with the opening words of the famous poem "The Road Not Taken" by Robert Frost: "Two roads diverged in a yellow wood, and sorry I could not travel both."

Not to be a poetry critic, but often there are more than two roads.

In Chapter 5, we evaluated only one Boolean expression at a time, and chose which of the two roads our code would travel down depending on whether the expression was true or false. However, sometimes two (or more) Boolean expressions need to be evaluated to determine the path the code will travel.

For example, you are eligible to vote only if you are a citizen *and* you are at least 18 years old. You cannot vote unless both conditions are true. Other times with Boolean expressions, you are testing if either of two comparisons is true.

For example, you may get into a movie free if you are either a senior citizen (65 years or older) *or* a child (12 or under). Thus, you get in free if either condition is true.

This chapter will cover two different approaches to evaluating two Boolean expressions to determine which code should execute. The first approach nests one if statement inside another. The second approach introduces another type of operator: logical operators.

Nested if Statements

An if statement may appear inside another if statement. When this is done, the inner if statement is said to be "nested" inside the outer if statement.

You can nest if statements to determine if both of two Boolean expressions are true, or if either of the expressions is true.

Testing if Both Boolean Expressions Are True

The following program shows the use of nested if statements in determining if both of two Boolean expressions are true. If the user's input is that they are at least 18 years old *and* a citizen, the program outputs that they are eligible to vote. Otherwise, the program outputs that they are not eligible to vote.

```
#include <iostream>
using namespace std;
int main(void)
{
   int age;
   char choice;
   bool citizen;
   cout << "Enter your age: ";
   cin >> age;
   cout << "Are you a citizen (Y/N): ";
   cin >> choice;
   if (choice == 'Y')
      citizen = true;
   else
      citizen = false;
   if (age >= 18)
      if(citizen == true)
         cout << "You are eligible to vote";
      else
```

```
         cout << "You are not eligible to vote";
   else
      cout << "You are not eligible to vote";
   return 0;
}
```

The following are several sample runs, each separated by ===:

```
Enter your age: 18
Are you a citizen (Y/N): Y
You are eligible to vote
===
Enter your age: 18
Are you a citizen (Y/N): N
You are not eligible to vote
===
Enter your age: 17
Are you a citizen (Y/N): Y
You are not eligible to vote
===
Enter your age: 17
Are you a citizen (Y/N): N
You are not eligible to vote
===
```

Figure 6-1 depicts a flowchart of this program.
The nested if portion of the program is

```
if (age >= 18)
   if(citizen == true)
      cout << "You are eligible to vote";
   else
      cout << "You are not eligible to vote";
else
   cout << "You are not eligible to vote";
```

Note: *The statement if(citizen == true) could be rewritten as if(citizen). The parentheses following the if keyword requires only an expression that evaluates to a Boolean value. Since citizen is a Boolean variable, it evaluates to a Boolean value without the need for any comparison.*

The if/else structure comparing whether the user is a citizen is nested within the if/else structure comparing whether the user is at least 18 years old. By this nesting, the comparison of whether the user is a citizen is made only if the user is at least

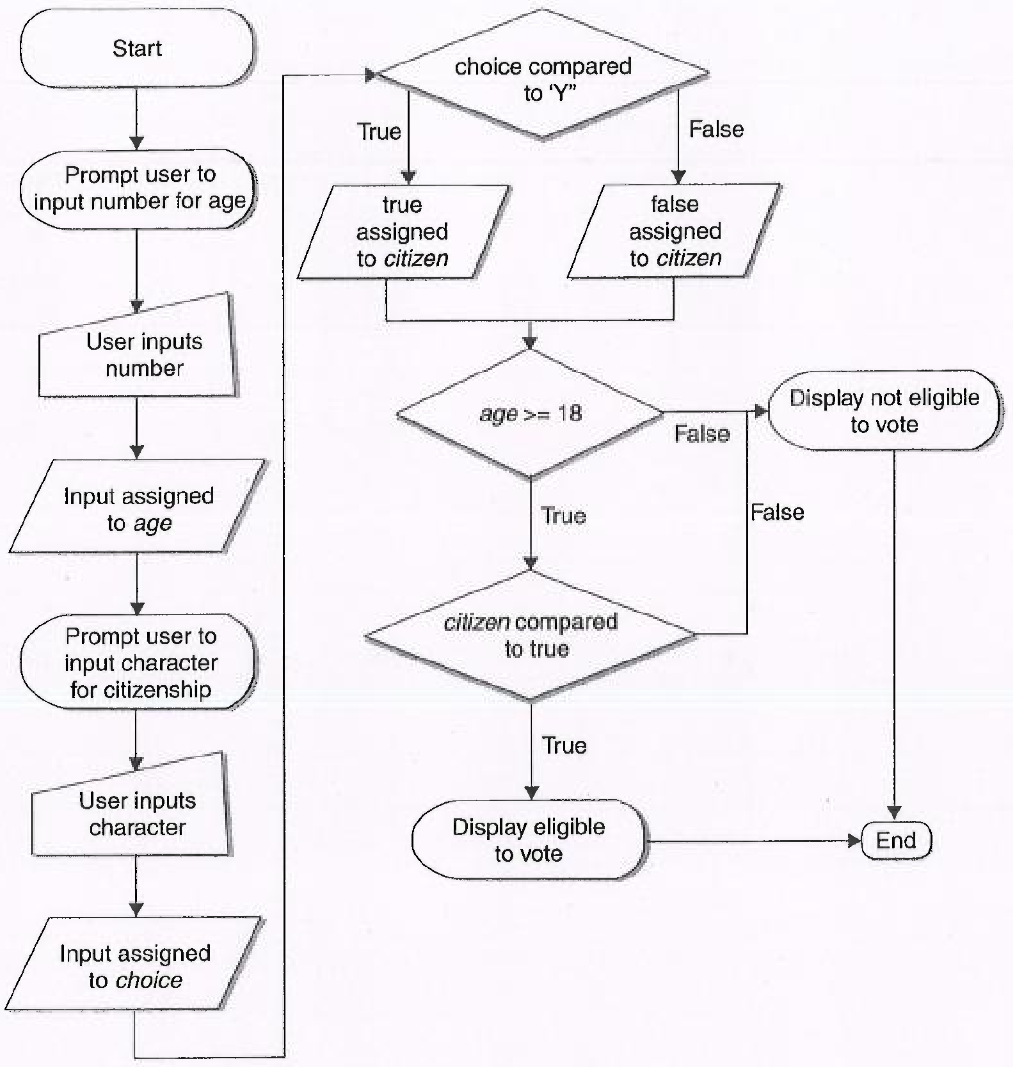

Start
Prompt user to input number for age
User inputs number
Input assigned to *age*
Prompt user to input character for citizenship
User inputs character
Input assigned to *choice*
choice compared to 'Y"
True
False
true assigned to *citizen*
false assigned to *citizen*
age >= 18
False
True
Display not eligible to vote
False
citizen compared to true
True
Display eligible to vote
End

Figure 6-1 Flowchart of the voting eligibility program

18 years old. This approach is logical, since if the user is not at least 18 years old, they will not be eligible to vote even if they are a citizen.

The if / else structure comparing whether the user is a citizen is referred to as the "inner" if / else structure. The if / else structure comparing whether the user is at least 18 years old is referred to as the "outer" if / else structure.

The entire inner if / else structure (comparing whether the user is a citizen) is nested within the if part of the outer if / else structure (comparing whether the user is at least 18 years old). You also can nest an if / else structure (or an if structure, or an if /else if /else structure) within the else if or else part of an outer if else/if else if else/if else structure.

This program illustrates a good use of nested if statements. It would be difficult to rewrite this program using an if / else if / else structure without nested if statements. However, later in this chapter we will cover another, equally good alternative: logical operators.

Testing if Either Boolean Expression Is True

The following program shows the use of the nested if statements in determining if either of two Boolean expressions are true. If the user's input indicates that they are either no more than 12 years old *or* at least 65 years old, the program outputs that their admission is free. Otherwise, the program outputs that they have to pay.

```
#include <iostream>
using namespace std;
int main(void)
{
   int age;
   cout << "Enter your age: ";
   cin >> age;
   if (age > 12)
      if (age >= 65)
         cout << "Admission is free";
      else
         cout << "You have to pay";
   else
      cout << "Admission is free";
   return 0;
}
```

The following shows several sample runs:

```
Enter your age: 12
Admission is free
===
Enter your age: 13
You have to pay
===
Enter your age: 65
Admission is free
```

Figure 6-2 depicts a flowchart of this program.

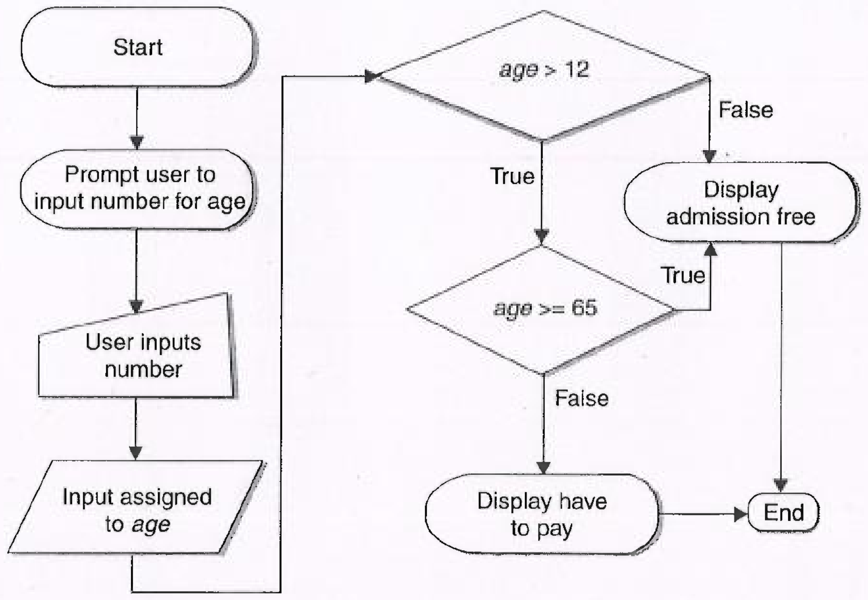


Figure 6-2 Flowchart of the movie admission program

The nested if portion of the program is

```
if (age > 12)
   if (age >= 65)
      cout << "Admission is free";
   else
      cout << "You have to pay";
else
   cout << "Admission is free";
```

The inner if/else structure, comparing whether the user is at least 65 years old, is nested within the outer if/else structure, comparing whether the user is over 12 years old. By this nesting, the comparison of whether the user is at least 65 years old is made only if the user is over 12 years old. This approach is logical, since if the user is no more than 12 years old, they will be admitted free (and also could not possibly be 65 years or older).

This program also could have been written using the following if / else if / else structure in place of the nested if statements:

```
if (age <= 12)
   cout << "Admission is free";
else if (age >= 65)
```

```
      cout << "Admission is free";
   else
      cout << "You have to pay";
```

Each of these two alternatives, the nested if statements and the if / else if / else structure, have disadvantages. Nesting one if statement inside another by its very nature may be somewhat difficult to write and understand. However, the if / elseif/else if / else structure has the disadvantage of repeating the same cout statement for both the if and else if parts. While this is just one line of repetitive code in this program, in more complex programs the repetitive code could be many lines long.

C++ has a third and perhaps better alternative, the use of logical operators, which we will discuss next.

Logical Operators

C++ has logical operators that enable you to combine comparisons in one if or else if statement. Table 6-1 lists the logical operators supported by C++ and describes what each does.

Operator	Name	What It Does
&&	And	Connects two relational expressions. Both expressions must be true for the overall expression to be true.
||	Or	Connects two relational expressions. If either expression is true, the overall expression is true.
!	Not	Reverses the "truth" of an expression, making a true expression false, and a false expression true.

Table 6-1 Logical Operators

The && Operator

The && operator also is known as the logical And operator. It is a binary operator; it takes two Boolean expressions as operands. It returns true only if both expressions are true. If either expression is false, the overall expression is false. Of course, if both expressions are false, the overall expression is false. Table 6-2 illustrates this.

Expression #1	Expression #2	Expression #1 && Expression #2
true	true	true
true	false	false
false	true	false
false	false	false

Table 6-2 The Logical And Operator

The following program shows the use of the logical And operator in determining whether the user is eligible to vote, the criteria being that the user must be at least 18 years old and a citizen.

```
#include <iostream>
using namespace std;
int main(void)
{
   int age;
   char choice;
   bool citizen;
   cout << "Enter your age: ";
   cin >> age;
   cout << "Are you a citizen (Y/N): ";
   cin >> choice;
   if (choice == 'Y')
      citizen = true;
   else
      citizen = false;
   if (age >= 18 && citizen == true)
      cout << "You are eligible to vote";
   else
      cout << "You are not eligible to vote";
   return 0;
}
```

The following are several sample runs, separated by ===:

```
Enter your age: 18
Are you a citizen (Y/N): Y
You are eligible to vote
===
Enter your age: 18
Are you a citizen (Y/N): N
You are not eligible to vote
===
Enter your age: 17
Are you a citizen (Y/N): Y
You are not eligible to vote
===
Enter your age: 17
Are you a citizen (Y/N): N
You are not eligible to vote
```

The part of the program that uses the logical And operator is

```
if (age >= 18 && citizen == true)
   cout << "You are eligible to vote";
else
   cout << "You are not eligible to vote";
```

The comparison *age >= 18* is referred to as the left part of the expression since it is to the left of the logical And operator. Similarly, the comparison *citizen == true* is referred to as the right part of the expression because it is to the right of the logical And operator.

If the user's age is at least 18 years, then the program makes the second comparison, whether the user is a citizen. If the user's age is not at least 18 years of age, the second comparison is not even made before the else part is executed. The reason is to avoid wasting CPU time, since if the left expression is false, the overall expression is false regardless of the result of the evaluation of the right expression.

Because the second comparison of whether the user is a citizen is made only if the user's age is at least 18, the flowchart in Figure 6-1 of this program using nested if statements also applies to this program using the logical And operator.

The || Operator

The || operator is also known as the logical Or operator. Like the logical And operator, the logical Or operator also is a binary operator, taking two Boolean expressions as operands. It returns true if either expression is true. It returns false only if both expressions are false. Of course, if both expressions are true, the overall expression is true. Table 6-3 illustrates this.

The following program shows the use of the logical Or operator in determining whether you get into a movie free, the criteria being that the user must be either no more than 12 or at least 65 years old.

Expression #1	**Expression #2**	**Expression #1 || Expression #2**
true	true	true
true	false	true
false	true	true
false	false	false

Table 6-3 The Logical Or Operator

```
#include <iostream>
using namespace std;
int main(void)
{
   int age;
   cout << "Enter your age: ";
   cin >> age;
   if (age <= 12 || age >= 65)
      cout << "Admission is free";
   else
      cout << "You have to pay";
   return 0;
}
```

The following shows several sample runs:

```
Enter your age: 12
Admission is free
===
Enter your age: 18
You have to pay
===
Enter your age: 65
Admission is free
```

The part of the program that uses the logical Or operator is

```
if (age <= 12 || age >= 65)
   cout << "Admission is free";
else
   cout << "You have to pay";
return 0;
```

As with the logical And operator, the comparison *age <= 12* is referred to as the left part of the expression and the comparison *age >= 65* is referred to as the right part of the expression.

If the user's age is over 12 years, then the program makes the second comparison, whether the user is at least 65 years of age. If the user is no more than 12 years of age, the second comparison is not even made before the else part is executed. The reason, as with the logical And operator, once again is to avoid wasting CPU time, since if the left expression is true, the overall expression is true regardless of the result of the evaluation of the right expression.

Because the second comparison of whether the user is at least 65 years old is made only if the user's age is over 12, the flowchart in Figure 6-2 of this program using nested if statements also applies to this program using the logical Or operator.

The ! Operator

The ! operator also is known as the logical Not operator. My daughters have been using the logical Not operator for years, telling me "Dad, you look just like Tom Cruise ... not!"

The logical Not operator inverts the value of the Boolean expression, returning false if the Boolean expression is true, and true if the Boolean expression is false. Table 6-4 illustrates this.

Expression	**!Expression**
true	true
false	true

Table 6-4 The Logical Not Operator

Unlike the logical And and Or operators, the logical Not operator is a unary operator; it takes only one Boolean expression, not two.

The following program shows the use of the logical Not operator, combined with the logical And operator, in determining whether you get into a movie for free.

```
#include <iostream>
using namespace std;
int main(void)
{
   int age;
   cout << "Enter your age: ";
   cin >> age;
   if (!(age > 12 && age < 65))
      cout << "Admission is free";
```

```
    else
       cout << "You have to pay";
    return 0;
}
```

This program is almost identical to the one used to illustrate the logical Or operator. The only difference is that the statement

```
    if (age <= 12 || age >= 65)
```

is replaced by the statement

```
    if (!(age > 12 && age < 65))
```

NOTE: *This change is an illustration of DeMorgan's law, which is a rule of inference pertaining to the logical And, Or, and Not operators that are used to distribute a negative to a conjunction or disjunction. In this book, it is only referred to and not covered, but in case you hear DeMorgan's law mentioned in a programming class or another book, you heard it here first!*

The Not operator permits you to state a Boolean expression a different way that may be more intuitive for you. In this example, expressing the condition for free admission as being that the age is not between 13 and 64 may be more intuitive than expressing that condition as being that the age is either no more than 12 or 65 or over.

Precedence

Table 6-5 lists precedence, from highest to lowest, among logical operators and between them and the relational operators.

Operator (from highest to lowest)
!
Relational operators (>, >=, <, <=, ==. !=)
&&
||

Table 6-5 The Precedence of Logical and Relational Operators

Precedence and the Logical Not Operator

Since the logical Not operator has a higher precedence than the relational operators, the program used to illustrate the logical Not operator uses an extra set of parentheses.

```
if (!(age > 12 && age < 65))
```

Had the extra set of parentheses been omitted as follows, the result would always be that the user has to pay. Thus, admission would never be free regardless of the age.

```
if (!age > 12 && age < 65)
```

The reason why the user always has to pay regardless of age is that since the logical Not operator has a higher precedence than the relational operators, the logical Not operator operates on *age,* not the expression *age > 12 && age < 65.* If *age* is non-zero, then *!age* is zero. Since 0 is not greater than 12, the left part of the logical And expression is false, so the overall expression is false.

The result of the user always having to pay regardless of age is the same even if *age* is zero. If *age* is zero, then *!age* is logical true, the integer equivalent of which usually is 1. Since 1 is not greater than 12, once again the left part of the logical And expression is false, so the overall expression is false.

Precedence and the Logical And and Or Operators

In contrast to the logical Not operator, the logical And and Or operators rank lower in precedence than the relational operators. Therefore, parentheses normally are not necessary to separate the logical And and Not operators from the relational operators. For example, the following two statements (the first taken from the program that illustrated the logical And operator) are equivalent.

```
if (age >= 18 && citizen == true)
if ( (age >= 18) && (citizen == true) )
```

However, parentheses are necessary when logical And and Or operators are used together in one statement and you want the Or done before the And since the logical And operator has higher precedence than the logical Or operator. This issue often arises when you have more than two Boolean expressions.

For example, assume the voting rules were changed so legal residents (represented by the Boolean variable *resident* having a value of true) as well as citizens who are at least 18 years old could vote. Given that assumption, the statement

```
if (resident == true || citizen == true && age >= 18)
```

would be the same as the following since the logical And operator has higher precedence than the logical Or operator.

```
if (resident == true || (citizen == true && age >= 18 ))
```

In this expression, a resident under 18 years old would be able to vote. The reason is that even if the expression (*citizen == true && age >= 18*) is false, as long as *resident* is true, the overall expression is true, since with a logical Or operator only one of the two Boolean expressions needs to be true for the overall expression to be true.

A resident under 18 years old being able to vote is not a correct result for this program. To avoid this logic error, parentheses would be necessary, so the logical Or operation is performed first.

```
if ( (resident == true || citizen == true) && age >= 18)
```

Using the switch Statement with Logical Operators

The switch statement was discussed at some length in Chapter 5. However, so far in this chapter it has been conspicuous by its absence.

In Chapter 5, we discussed how the switch statement was cumbersome when dealing with a range of numbers. The reason was that the case keyword cannot be followed by a range of numbers because it must instead be followed by a single integer constant.

However, the switch statement may be used with expressions that use the logical And or Or operator. The reason is that these expressions have only one of two possible values, true or false. True and false are both constants; the value of true is always true and the value of false is always false. While true and false are Boolean values, each has a corresponding integer value: 1 and 0. Therefore, the case keyword may be followed by true or false, just as in Chapter 5 where the case keyword can be followed by a character since a character has a corresponding integer ANSI or ASCII value.

For example, earlier in this chapter the logical And operator was used in the following if/else structure in determining whether the user is eligible to vote, the criteria being that the user must be at least 18 years old and a citizen.

```
if (age >= 18 && citizen == true)
   cout << "You are eligible to vote";
else
   cout << "You are not eligible to vote";
```

The corresponding switch statement is

```
   switch (age >= 18 && citizen == true)
   {
   case true:
      cout << "You are eligible to vote";
      break;
   case false:
      cout << "You are not eligible to vote";
}
```

Also earlier in this chapter, the logical Or operator was used in the following if/else structure in determining whether the user gets into a movie free, the criteria being that the user must be either under 18 or at least 65 years old.

```
   if (age <= 12 || age >= 65)
      cout << "Admission is free";
   else
      cout << "You have to pay";
```

The corresponding switch statement is

```
switch (age <= 12 || age >= 65)
{
   case true:
      cout << "Admission is free";
      break;
   case false:
      cout << "You have to pay";
}
```

These examples illustrate that the switch statement can be employed as an alternative to an if/else or if/else if/else structure in programs that evaluate Boolean expressions using logical operators. However, it is not common for the switch statement to be employed in this manner because, with Boolean expressions, there are always just two alternatives, true and false, and switch statements generally are used when there are many more alternatives than two.

Summary

In Chapter 5, we evaluated only one Boolean expression at a time to determine which of two alternative blocks of code should execute. However, often two (or more) Boolean expressions need to be evaluated to determine which block of code should execute. In the example in which you are eligible to vote only if the user is a citizen *and* at least 18 years old, both Boolean expressions must be true in order for the program to output that the user is eligible to vote. In another example, in which you get into a movie free if the user is either a senior citizen (65 years or older) *or* a child (12 or under), the program outputs that the user gets into the movie free if either Boolean expression is true.

This chapter covered two different approaches of evaluating two Boolean expressions to determine which code should execute. The first approach nested one if statement inside another. The second approach introduced three logical operators. The logical && (And) operator is used when both Boolean expressions must be true. The logical || (Or) operator is used when either Boolean expression must be true. Finally, the logical ! (Not) operator inverts the value of a Boolean expression, from true to false, or false to true.

Finally, this chapter showed how you can use the switch statement as an alternative to an if / else or if / else if /else structure in programs that evaluate Boolean expressions using logical operators.

Quiz

1. Can you use nested if statements as an alternative to the logical And and Or operators?
2. Can an if statement be nested in the else if or else part of an if / else if / else statement, or just the if part?
3. For which of the logical operators do both Boolean expressions have to be true for the overall Boolean expression to be true?
4. For which of the logical operators do both Boolean expressions have to be false for the overall Boolean expression to be false?
5. Which of the logical operators reverses the "truth" of a Boolean expression, making a true expression false and a false expression true?
6. Assuming *resident* is a Boolean variable, is *if(resident)* the same as *if(resident == true)?*
7. Which of the logical operators is a unary rather than binary operator?
8. Which of the logical operators has a higher precedence than the relational operators?
9. Which logical operator has a higher precedence, And or Or?
10. Can a Boolean value of either true or false be used following the case keyword in a switch statement?

CHAPTER 8

While and Do While Loops

The for loop generally is used when the loop will iterate a fixed number of times. However, sometimes the number of times a loop will iterate is unpredictable, depending on user input during runtime. For example, in a data entry application, you may want a loop that, upon entry of invalid data, asks the user whether they want to retry or quit, and if they want to retry, gives the user another opportunity to enter data. The number of times this loop may iterate is unpredictable, since it will keep repeating until the user either enters valid data or quits.

This chapter will show you how to use the while loop, which is a better choice than a for loop when the number of times a loop will iterate is unpredictable.

While the total number of loop iterations may be unpredictable, there often are situations in which the loop will iterate at least once. An example is a loop that displays a menu with various choices, including exiting the program. In this menu example, the menu always displays at least once; the user cannot choose to exit before being given that choice. In such situations, a do while loop, which this chapter will show you how to use, is a better choice than a while loop.

The While Loop

The while loop is similar to a for loop in that both have the typical characteristics of a loop: the code inside each continues to iterate until a condition becomes false. The difference between them is in the parentheses following the for and while keywords.

The parentheses following the for keyword consists of three expressions, initialization, condition, and update. By contrast, the parentheses following the while keyword consists only of the condition; you have to take care of any initialization and update elsewhere in the code.

This difference is illustrated by the following program that outputs the numbers between 1 and 10. Chapter 7 included the following program that outputs the numbers between 1 and 10 using the for loop.

```
#include <iostream>
using namespace std;
int main(void)
{
   for (int num = 1; num <= 10; num++)
      cout << num << " ";
   return 0;
}
```

The same program using the while loop could be

```
#include <iostream>
using namespace std;
int main(void)
{
   int num = 1;
   while (num <= 10)
   {
      cout << num << " ";
      num++;
   }
   return 0;
}
```

Note: The two statements in the body of the while loop could have been combined into one statement, cout << num++. Two statements are used instead to make this example easier to understand by eliminating the precedence issue in the one statement between the stream insertion and increment operators.

With the while loop, the integer variable *num* had to be declared and initialized before the loop since this cannot be done inside the parentheses following the while keyword. Further, *num* was updated inside the code of the loop using the increment operator. This update also can be done inside the parentheses following the while keyword as shown by an example later in this section.

The update of the variable is particularly important with the while loop. Without that update, the loop would be infinite. For example, in the following excerpt from this program, if num is not incremented, the loop would be infinite. The value of num would not change from 1, so the condition *num <= 10* always would remain true.

```
int num = 1;
while (num <= 10)
   cout << num << " ";
```

Forgetting to update the value of the variable you are using in the condition is a common mistake with a while loop. Forgetting the update is less common with a for loop because that update is the usual purpose of the third expression in the paren theses following the for keyword.

Otherwise, the syntax rules discussed in Chapter 7 concerning the for loop apply equally to the while loop. For example, if more than one statement belongs to the while loop, then the statements must be contained within curly braces. That is why in the program that outputs the numbers between 1 and 10 using the while loop, the two statements in the body of the while loop are contained within curly braces.

```
while (num <= 10)
{
   cout << num << " ";
   num++;
}
```

In the program we just analyzed, the update of the value of *num* was done within the body of the loop. The update could also be done within the condition itself:

```
#include <iostream>
using namespace std;
int main(void)
{
   int num = 0;
   while (num++ < 10)
      cout << num << " ";
   return 0;
}
```

Updating the counter within the condition requires two changes from the previous code. First, the value of *num* has to be initialized to 0 instead of to 1 because the increment inside the parentheses during the first iteration of the loop would change that variable's value to 1. Second, the relational operator in the condition is < rather than <= because the value of *num* is being incremented before it is outputted.

Updating the counter within the condition raises the question: Given the condition *num++ < 10,* which comes first, the comparison or the increment? Since the increment is postfix, the answer is the comparison.

The counter also could be updated within the condition using a prefix increment. However, then the condition should be *++num <= 10* to obtain the desired output.

As with the for loop, the statement or statements following the while keyword and parentheses will not execute if the parentheses is followed by a semicolon, as that would be interpreted as an empty statement. Test yourself on this; what would be the output if we placed a semicolon after the while condition as in the following code fragment?

```
while (num <= 10);
   cout << num++ << " ";
```

The only number that would output is 11. The reason is that the loop continues, and the empty statement executes, until the condition fails when *num* is 11, at which time the statement following the loop executes and the value of *num* (11) is outputted.

Comparison of for and while Loops

The practical difference between the for and while loops is not apparent in a program with a predictable number of iterations, such as the program we have been discussing thus far that outputs the numbers between 1 and 10. Rather, a while loop is a superior choice to a for loop in a program where the number of iterations is unpredictable, depending on user input during runtime.

For example, in the following program, the program asks the user to enter a positive number, and in a loop continues that request until the user does so. The number of times this loop may execute is unpredictable. It may never execute if the user enters a positive number the first time, or it may execute many times if it takes the user several tries to enter a positive number.

```
#include <iostream>
using namespace std;
int main(void)
{
   int num;
   cout << "Enter a positive number: ";
   cin >> num;
```

```
    while (num <= 0)
    {
        cout << "Number must be positive; please retry: ";
        cin >> num;
    }
    cout << "The number you entered is " << num << " ";
    return 0;
}
```

Here is some sample input and output:

```
Enter a positive number: 0
Number must be positive; please retry: -1
Number must be positive; please retry: 3
The number you entered is 3
```

This program would be more difficult to write with a for loop. While it could be done, the for loop is designed for situations in which the number of iterations is predictable.

Using the break Keyword

Even though the while loop is a better choice than a for loop for this program, which requires the user to enter a positive number, there are two problems with this program: one minor and one major.

The minor problem is that there is some repetition of code; the user is requested both before and inside the loop to enter a positive number. A do while loop, which is explained in the following section, avoids this repetition, but repeats other code (there are tradeoffs in loops as well as in life).

The major problem is that the user is trapped inside the loop until they enter a positive number. That is not a good programming design. While the user should be required to enter good data if they are going to enter any data at all, they should have the option, when told the data entered was not valid, of quitting the data entry.

The following modification of the program uses the break keyword to provide the user with the option of quitting the data entry:

```
#include <iostream>
using namespace std;
int main(void)
{
    int num;
    char choice;
    cout << "Enter a positive number: ";
    cin >> num;
    while (num <= 0)
    {
        cout << "Number must be positive; try again (Y/N): ";
        cin >> choice;
        if (choice == 'Y')
        {
            cout << "Enter number: ";
            cin >> num;
        }
        else
            break;
    }
    cout << "The number you entered is " << num << " ";
    return 0;
}
```

Here is some sample input and output when the user eventually enters a positive number:

```
Enter a positive number: 0
Number must be positive; try again (Y/N): Y
Enter number: -1
Number must be positive; try again (Y/N): Y
Enter number: 3
The number you entered is 3
```

Here is some sample input and output when the user does not enter a positive number but instead decides to quit:

```
Enter a positive number: -2
Number must be positive; try again (Y/N): N
The number you entered is -2
```

Flags

The flags modification is an improvement because the user no longer is trapped inside the loop until they enter a positive number, but instead has the option of quitting data entry. However, the second sample input and output, in which the user quits data entry, illustrates a problem. The final cout statement outputs the number entered, even if the number is invalid data.

Ideally, we would only want to output the data if it were valid. If the data were not valid, then we would want to output that fact instead. However, the code thus far does not enable us to differentiate whether the while loop ended because the user entered valid data or because the user decided to quit after entering invalid data.

In Chapter 7, I recommended that you use the break keyword sparingly because it created multiple exit points for the for loop, making your code more difficult to understand and increasing the possibility of logic errors. That advice also applies to the while loop. I recommended then, and recommend now, as an alternative the use of a logical operator. The following program modification adopts that alternative.

```
#include <iostream>
using namespace std;
int main(void)
{
   int num;
   char choice;
   bool quit = false;
   cout << "Enter a positive number: ";
   cin >> num;
   while (num <= 0 && quit == false)
   {
      cout << "Number must be positive; try again (Y/N): ";
      cin >> choice;
      if (choice != 'Y')
      {
         cout << "Enter number: ";
         cin >> num;
      }
      else
         quit = true;
   }
   if (quit == false)
      cout << "The number you entered is " << num << " ";
   else
      cout << "You did not enter a positive number";
   return 0;
}
```

Here is some sample input and output when the user eventually enters a positive number:

```
Enter a positive number: -3
Number must be positive; try again (Y/N): Y
Enter number: 3
The number you entered is 3
```

Here is some sample input and output when the user does not enter a positive number but instead decides to quit. This time the final output is not of the number entered, but rather that the user did not enter a positive number:

```
Enter a positive number: 0
Number must be positive; try again (Y/N): Y
Enter number: -1
Number must be positive; try again (Y/N): N
You did not enter a positive number
```

This program modification, in addition to using the logical && operator, uses a Boolean variable named *quit*. This Boolean variable is used as a *flag*. A flag is a Boolean variable whose value indicates whether a condition exists.

In this program, the while loop continues to loop as long as the data entered is invalid *and* the user wants to keep going. Accordingly, the while keyword is followed by two conditions, joined by the logical && operator.

NOTE: *A common programming mistake in a while condition using a logical operator is to use && when you should use || or vice versa. While the logical && operator may seem the obvious choice in this example, the correct choice in other situations may be less intuitive. For example, if you want to loop while a number is not between 1 and 10, would the loop be while (num < 1 && num > 10) or while (num < 0 || num > 10)? The answer is the latter; the condition always would be false using the && operator since a number cannot be both less than 1 and greater than 10. If you wanted to use the && operator, the condition instead would be while (num >= 1 && num <= 10).*

The first condition is if *num <= 0*. If this expression is false, the data is valid, so the issue of whether the user wants to quit does not arise. Accordingly, the second condition, whether *quit* is true, is not even evaluated. As discussed in Chapter 7, with a logical && operator, the right expression is evaluated only if the left expression is true. Therefore, the while loop ends with the value of *quit* being false, its initialized value, and code execution continues with the if / else statement following the while loop.

However, if *num <= 0* is true, then the data is invalid, and the second condition, whether *quit* is true, is evaluated.

The value of *quit* may be true under either of two possibilities. The first possibility is that this is the user's first attempt to enter data and the data was invalid. In this case, the user has not yet been asked whether they want to quit. It is assumed they don't, so they have the opportunity to answer whether they want to retry. Therefore, the *quit* variable is initialized to the value of false when it is declared.

The second possibility is that this is the user's second or later attempt to enter data and the data was invalid. In this case, the user has already been asked whether they want to quit, so the value of quit is based on the user's answer.

If the value of *quit* is false, the while loop continues. However, if the user wants to quit, then the right expression *quit == false* will be false because the value of *quit* is true. Therefore, the while loop ends with the value of *quit* being true, and code execution continues with the if / else statement following the while loop.

At some point (hopefully) the while loop will end, either because the user has entered a valid number or has not and decided to quit trying. Code execution then continues with the if / else statement following the while loop.

The value of *quit* being false necessarily indicates that the user entered valid data, because if they were still trying to do so, the loop would not have ended. Conversely, the value of *quit* being true necessarily indicates that the user entered invalid data.

Accordingly, we use the value of *quit* in the if /else statement after the while loop to differentiate whether the while loop ended because the user entered valid data or instead decided to quit after entering invalid data.

Thus, inside the while loop, *quit* is a flag whose value indicates whether the user wants to try again, and after the while loop ends, *quit* is a flag whose value indicates whether the user entered valid data.

While (true)

In Chapter 7, we discussed the use of the for loop with the omission of the condition that is the second expression, such as *for (; ;)*. There, an infinite loop was avoided by using the break keyword inside the loop. While I did not recommend this use of the for loop, I mentioned it because you may encounter it as programmers do use the for loop this way.

Similarly, programmers sometimes make the condition of the while loop always true, such as while (true) or while (1), and break out of the while loop with, you guessed it, the break keyword. Here is an example that is a modification of the program we have been using that asks the user to enter a positive number.

```
#include <iostream>
using namespace std;
int main(void)
{
   int num;
   char choice;
   bool quit = false;
   while (true)
   {
      cout << "Enter a positive number: ";
      cin >> num;
      if (num > 0)
```

```
        break;
        else
        {
        cout << "Number must be positive; try again (Y/N): ";
        cin >> choice;
        if (choice != 'Y')
        {
        quit = true;
        break;
        }
        }
    }
    if (quit == false)
        cout << "The number you entered is " << num << " ";
    else
        cout << "You did not enter a positive number";
    return 0;
}
```

The one advantage of this modification is that it renders unnecessary having to prompt the user both before and inside the loop to enter a positive number. However, the use of the *while (true)* syntax has the disadvantage of making your code less readable because the condition that stops the loop cannot be discerned from the parentheses following the while keyword. The do while loop (explained later in this chapter) avoids this disadvantage and would be a preferable choice.

The continue Keyword

You can use the continue keyword in a while loop just as you can in a for loop. As discussed in Chapter 7, the continue keyword, like the break keyword, is used within the code of a loop, commonly within an if / else structure. If the continue statement is reached, the current iteration of the loop ends, and the next iteration of the loop begins.

Chapter 7 demonstrated the use of the continue keyword in a program in which the user is charged $3 an item, but not charged for a "baker's dozen," so every 13th item is free—that is, the user is only charged the price for a dozen items, even though they receive 13. The following is a modification of that program using a while loop.

```
#include <iostream>
using namespace std;
int main(void)
```

```
{
   int num, counter = 0, total = 0;
   cout << "How many items do you want to buy: ";
   cin >> num;
   while (counter++ < num)
   {
      if (counter % 13 == 0)
         continue;
      total += 3;
   }
   cout << "Total for " << num << " items is $" << total;
   return 0;
}
```

Note: The % (modulus) operator is used if the remainder is 0, 13, or a multiple of 13 items.

While this use of the continue keyword certainly works, as I cautioned in Chapter 7, you should use it (as well as the break keyword) sparingly. Normally, each iteration of a for loop has one end point. However, when you use a continue statement, each iteration has multiple end points. This makes your code more difficult to understand, and can result in logic errors.

I suggested in Chapter 7, in an example using the for loop, that you could use the logical ! (Not) operator as an alternative to using the continue keyword. Here is how you could do so using the while loop.

```
#include <iostream>
using namespace std;
int main(void)
{
   int num, counter = 0, total = 0;
   cout << "How many items do you want to buy: ";
   cin >> num;
   bool keepgoing = true;
   while (counter++ < num)
   {
      if (! (counter % 13 == 0 ))
         total += 3;
   }
   cout << "Total for " << num << " items is $" << total;
   return 0;
}
```

Note: *You also could use the relational != (not equal) operator, changing the if statement to if (counter % 13 != 0).*

Nesting While Loops

In Chapter 7, I showed you how you can nest one for loop inside another. Similarly, you can nest one while loop inside another. You also can nest a while loop inside of a for loop, or a for loop inside of a while loop.

Chapter 7 demonstrated nested for loops with a program that prints 5 rows of 10 X characters. The following is a modification of that program using nested while loops.

```
#include <iostream>
using namespace std;
int main(void)
{
   int x = 0;
   while (x++ < 5)
   {
      int y = 0;
      while (y++ < 5)
         cout << "X";
      cout << '\n';
   }
   return 0;
}
```

The variable *y*, used as a counter in the inner while loop, needs to be reinitialized in the outer while loop. The variable *y* could be declared outside the loops, but it needs to be assigned (or reassigned) the value of zero inside the outer loop since the inner loop goes through all of its iterations for each iteration of the outer loop.

Since each loop has a predictable number of iterations, using nested for loops is somewhat simpler than using nested while loops. However, both work.

The Do While Loop

The do while loop is similar to the while loop. The primary difference is that with a do while loop the condition is tested at the bottom of the loop, unlike a while loop where the condition is tested at the top. This means that a do while loop will always execute at least once, whereas a while loop may never execute at all if its condition is false at the outset.

Syntax

The syntax of a do while loop is

```
do {
   statement(s);
} while (condition);
```

The do keyword starts the loop. The statement or statements belonging to the loop are enclosed in curly braces. After the close curly brace, the while keyword appears, followed by the condition in parentheses, terminated by a semicolon.

A Do While Loop Example

The following program is a modification of the one earlier in this chapter that used a while loop to continue to prompt the user to enter a positive number until the user either did so or quit, and then either outputted the positive number or a message that the user did not enter a positive number. This modification uses a do while loop instead of a while loop.

```
#include <iostream>
using namespace std;
int main(void)
{
   int num;
   char choice;
   bool quit = false;
   do {
      cout << "Enter a positive number: ";
      cin >> num;
      if (num <= 0)
      {
      cout << "Number must be positive; try again (Y/N): ";
      cin >> choice;
      if (choice != 'Y')
      quit = true;
      }
   } while (num <= 0 && quit == false);
   if (quit == false)
      cout << "The number you entered is " << num << " ";
   else
      cout << "You did not enter a positive number";
   return 0;
}
```

The following are sample inputs and outputs. The first one has the user successfully enter a positive number the first time.

```
Enter a positive number: 4
The number you entered is 4
```

The next sample input and output has the user enter a positive number after two unsuccessful tries.

```
Enter a positive number: 0
Number must be positive; try again (Y/N): Y
Enter a positive number: -1
Number must be positive; try again (Y/N): Y
Enter a positive number: 4
The number you entered is 4
```

The final sample input and output has the user quit after two unsuccessful tries.

```
Enter a positive number: 0
Number must be positive; try again (Y/N): Y
Enter a positive number: -1
Number must be positive; try again (Y/N): N
You did not enter a positive number
```

Comparison of the Do While and While Loop

The preceding program, which used the do while loop, did not need to prompt the user both before and inside the loop to enter a number as did the corresponding program that used the while loop. However, this program using the do while loop repeats the *num* <= *0* condition inside the loop, whereas the corresponding program that used the while loop did not need to do that.

As a general rule, I prefer a do while loop over a while loop in those situations in which the loop must execute at least once before a condition may be tested, simply because under these circumstances it seems illogical to test the condition prematurely on the first iteration of the loop. As you may recall, in the program variation that used the while loop, the value of *quit* could be true in the loop condition under either of two possibilities, one being it was the user's first attempt to enter data so the user has not yet been asked whether they want to quit, and the other being it was the user's second or later attempt to enter data and the user answered that they wanted to quit. By contrast, using the do while loop eliminates the first possibility.

The preceding program, in which the user had to enter a number, whether that number is positive or not, is an example of the situation in which the loop must execute at least once before a condition may be tested. Another common example of this situation is when a menu is displayed. Assume the program displays a menu such as the following:

```
Menu
====
1. Add an entry
2. Edit an entry
3. Delete an entry
4. Exit
```

If the user chooses options 1, 2, or 3, the program performs the indicated operation (add, edit, or delete) and then again displays the menu for the user's next choice. If the user chooses option 4, the program ends.

In this menu example, the menu always displays at least once; the user cannot choose to exit before being given that choice. Accordingly, a do while loop normally is preferable to a while loop when choosing a loop to display a menu.

Scope

With a do while loop, it is important that a variable used in the condition following the while keyword not be declared inside the loop.

In the program that demonstrated the do while loop, the variables num and *quit* were declared before the loop:

```
int num;
char choice;
bool quit = false;
do {
   // statements
} while (num <= 0 && quit == false);
```

These variables could not be declared inside the do while loop, as in the following code excerpt, because the code would not compile. The parentheses following the while keyword is highlighted, and the compiler error is that *num* and *quit* are undeclared identifiers.

```
char choice;
do {
   int num;
   bool quit = false;
   // more statements
} while (num <= 0 && quit == false);
```

The reason why this alternative will not compile concerns variable *scope.*

As you know from Chapter 3, a variable must be declared before it can be referred to in code. Once a variable is declared, it may be referred to wherever in the code it has scope.

Thus far, variables have been declared in main, just after the open curly brace which begins the body of the main function. This gives these variables scope until the close curly brace, which ends the body of the main function. Since thus far our programs have had only one function, main, as a practical matter, the variables, once declared, could be referred to throughout the entire program.

In this example, however, the variables *num* and *quit* are declared after the open curly brace that begins the body of the do while loop. That means their scope is limited to the area between that open curly brace and the close curly brace that ends the body of the do while loop. This area between an open and close curly brace also is referred to as a *block.*

The while keyword and the parentheses that follow it are outside the body of the do while loop, or put another way, after the close curly brace that ends the body of the do while loop. Since the variables *num* and *quit* were declared within the body of the do while loop, they do not have scope outside the body of the loop where the while parentheses are located. Therefore, these variables are regarded as undeclared when referred to within those parentheses.

This issue arises far more often with the do while loop than with the for or while loops. With for or while loops, the condition precedes the body of the loop, so any variables used in the condition necessarily would be declared before the loop or, in the case of the for loop, within the parentheses following the for keyword. By contrast, since the condition of a do while loop comes after the body of the loop, it is an easy mistake to declare the variables used in the condition before it, in the body of the loop.

This is our first discussion of the variable scope issue. However, it is by no means our last. This issue is not limited to the do while loop. It arises frequently when we start adding other functions to our programs, as we will do in upcoming chapters.

Summary

Chapter 7 introduced the first of several loops: the for loop. The for loop works well in situations where the loop will iterate a fixed number of times.

Often, however, the number of times a loop will iterate is unpredictable since the number of iterations depends on user input during runtime. One example discussed in this chapter is a data entry application in which the loop, upon entry of invalid data, asks the user whether they want to retry or quit, and if they want to retry, gives the user another opportunity to enter data. The number of times this loop may iterate is unpredictable, since it will keep repeating until the user either enters valid data or quits.

This chapter showed you how to use the while loop, which works better than a for loop when the number of times a loop will execute is unpredictable. While the parentheses following the for keyword consists of three expressions, initialization, condition, and update, the parentheses following the while keyword consists only of the condition; you have to take care of initialization and update elsewhere in the code.

There also are situations in which, while the number of times this loop may execute is unpredictable, the loop will execute at least once. An example discussed in this chapter is a loop that displays a menu with various choices, including exiting the program. In this menu example, the menu always displays at least once; the user cannot choose to exit before being given that choice. In such situations, a do while loop is a better choice than a while loop. This chapter showed you how to use a do while loop, and introduced the issue of variable scope.

So far, all of our programs have had only one function, main. While all programs must have a main function, a C++ program may have additional functions. As programs get more sophisticated, it is helpful not to put all the code in main, but instead to allocate the code among different functions. The next chapter will show you how to add and use additional functions.

Quiz

1. Which of the three loops—for, while, or do while—executes at least once?
2. Which of the three loops—for, while, or do while—is the best choice when the number of iterations is predictable?
3. Is the parenthetical expression following the while keyword for initialization, condition or update?
4. May the parenthetical expression following the while keyword be true, such as *while (true)?*
5. Can the parenthetical expression following the while keyword combine two expressions?
6. What is the purpose of the break keyword in a while loop?
7. What is the purpose of the continue keyword in a while loop?
8. What is a flag?
9. If you were going to use nested while loops to print rows and columns, which for loop would print the rows, inner or outer?
10. Does a variable declared inside the body of a do while loop have scope in the parenthetical expression following the while keyword?

CHAPTER 13

Persistent Data: File Input and Output

As a kid, I had to listen patiently, or not so patiently, to endless (so it seemed at the time) lectures from my parents on how I could be better, or do better. After I became an adult, I realized to my amazement that my parents more often than not were right. Indeed, after I became a parent, I realized to my horror that I was repeating their lectures to my own children, who, of course, today enjoy these "talks" about as much I used to.

One of my parents' favorite lectures was about how important it is to be persistent. Once again, mom and dad showed true insight, because, though persistence is a very valuable trait in any person, it is particularly important in programmers.

Data, as well as programmers, should be persistent. By persistent, I mean the data should survive when the program is finished. Can you imagine if, after typing this chapter, when I exited Microsoft Word, everything I typed was lost?

With the programs we have written so far, this is exactly what would happen. Whatever values we have stored in variables do not persist, or survive, when the program is finished. Instead, the data is lost because the data is stored in RAM (random access memory), which is cleared when the program (or the computer) stops running.

Fortunately, Microsoft Word (and most programs for that matter) has the capability to save data to a file on the computer's hard drive or other storage medium so that data later can be retrieved when needed. That data persists after the termination of the program or even after the computer is turned off.

This chapter will show you how to make your data persistent by saving it to a file. Of course, saving the data accomplishes little unless you can later retrieve it, so this chapter also will show you how to retrieve data from a file.

Text vs. Binary Files

If you work on a computer, you work with files. You may have worked with hundreds if not thousands of files. However, have you ever stopped to think about what a file exactly is?

A file is a collection of data, and is located on persistent storage (discussed in Chapter 2) such as a hard drive, a CD-ROM, or other storage device.

A file is referred to by a name (called, naturally enough, a filename), which often is descriptive of the nature or contents of the file. For example, the Microsoft Word document for this chapter may be named *chapter13*.

A filename usually has an extension, beginning with a period (.). For example, if the file for this chapter is named *chapter13.doc,* the extension is *.doc.*

The purpose of the file extension is to indicate the type of date in the file and the program that normally is used to access the file. Accordingly, by convention, *.doc* is the extension for files normally accessed by Microsoft Word, *.xls* is the extension for files normally accessed by Microsoft Excel, and so forth. One extension you may have used frequently when working with this book is *.cpp,* for C++ source files.

As there are many types of programs, there are many types of files, and many different file extensions. However, fundamentally, there are two types of files: text and binary.

A text file is, as the name suggests, a file that contains text. An example is a file you might create in Notepad or another plain-text editor.

The meaning of binary in a binary file is less intuitive. View a Microsoft Word document in Notepad or another plain-text editor, such as the one I used to type this chapter. You will see, in addition to the text, strange characters such as ã6, ÌL, h5, and dark vertical lines that most definitely do not appear in the text. These are formatting codes used by Microsoft Word to format the text, such as for tables, bulleted and numbered lists, and so forth.

Text files can only store text. By contrast, binary files can store other types of information, such as images, database records, executable programs, and so forth. Consequently, more complex programs, such as Microsoft Word, Excel, or Access, store data in binary files.

Text files are somewhat simpler than binary files to access, read, and write. Consequently, file access usually is introduced using text files, with binary files a more advanced topic. This being an introductory-level book, I will use text files when explaining file access. However, when pertinent during this chapter, I also will refer to binary files.

The fstream Standard Library

We have been using the *iostream* standard library, which supports, among other functionalities, *cin* for reading from standard input (usually the keyboard), and *cout* for outputting to standard output (usually the monitor).

Reading or writing from a file requires another standard library, *fstream.* The fstream standard library is included with the statement:

```
#include <fstream>
```

Both *iostream* and *fstream* have in common the word "stream." This is no accident. Both standard libraries concern streams of bytes. The *iostream* library concerns streams of bytes resulting from the "io" in *iostream,* input and output. The *fstream* standard library concerns streams of bytes resulting from the "f" in *fstream,* a file.

The *fstream* header file defines three new data types:

- *ofstream* This data type represents the output file stream—the "o" in *ofstream* standing for output. The direction of output is from your program out to a file. The *ofstream* data type is used to create files and to write information to files. It cannot be used to read files.
- *ifstream* This data type represents the input file stream—the "i" in *ifstream* standing for input. The direction of input is from a file into your program.

The *ifstream* data type is used to read information from files. It cannot be used to create files or to write information to them.

- *fstream* This data type represents the file stream generally, and has the capabilities of both *ofstream* and *ifstream*. It can create files, write information to files, and read information from files.

The File Access Life Cycle

When your program accesses a file, whether to read it, write to it, or both, it goes through the following steps.

The file first must be opened. This establishes a path of communication between the file and a stream object in your program—*fstream, ofstream,* or *ifstream*—used to access the file.

Your program then reads from, or writes to, the file (or both). This section will discuss writing to a file before reading to it, but in your program the order could be reversed. Additionally, your program may only read from a file, or only write to a file.

Finally, your program closes the file. Maintaining the path of communication between the file and the stream object in your program requires system resources, so closing the file frees those resources when they are no longer needed. Additionally, you may not be able to access the file later in your program if you did not close it after the previous access.

Opening a File

A file must be opened before you can read from it or write to it. As discussed in the introduction to this section, opening a file establishes a path of communication between the file and a stream object in your program. Opening a file for writing is first discussed.

Opening a File for Writing

Either the *ofstream* or *fstream* object may be used to open a file for writing. However, the *ifstream* object cannot be used for this purpose because it only may be used to read from a file.

Both the *ofstream* and *fstream* objects may open a file one of two ways. The first way is using a member function named, as you might expect, *open*. The second alternative is using a constructor, which is explained in the "The *fstream* or *ofstream* Constructor" section later in this chapter.

The *Open* Member Function

Both the *ofstream* and *fstream* objects use an *open* member function, whose first argument is the name and location of the file to be opened. However, whether you include a second argument may depend on whether the *ofstream* or *fstream* object is calling the *open* member function, or whether you want to access the file in a different "mode" than the default.

First Argument—Specifying the File to Be Opened

The file to be opened for writing need not already exist. If it does not, attempting to open it for writing to it automatically will create it with the specified name at the specified location. However, whether or not the file yet exists, you need to specify a file name and location.

Accordingly, whether the *ofstream* or *fstream* object is calling the function, the first argument specifies the name and location of the file to be opened. This information may be provided by using either the *relative path* or *absolute path* of the file. The terms *relative path* and *absolute path* are new, so let's discuss them now.

The relative path is the path relative to the location of your program. For example, the following statements open for writing a file, *students.dat,* that is in the same directory as the program:

```
ofstream outfile;
outfile.open("students.dat");
```

By contrast, the absolute path is the path starting with the drive letter, and including each directory and subdirectory until the file is reached. For example, if the *students.dat* file is in the *Classes* subdirectory of the *College* directory of my C drive, it would be opened for writing, using the absolute path, as follows:

```
ofstream outfile;
outfile.open("c:\\college\classes\\students.dat");
```

NOTE: *Two backslashes are necessary because one backslash is used to note an escape sequence. Two backslashes is the escape sequence for one backslash.*

Whether you use a relative or absolute path, the argument for the *open* function need not be a string literal. It also may be a string variable, as in the following code fragment:

```
ofstream outfile;
char filename[80];
cout << "Enter name of file: ";
cin >> filename;
outfile.open(filename);
```

NOTE: *As a general rule, using a relative path is preferable, particularly if the program will be used on different machines. While the location of the data file relative to the program directory may remain the same, there is no guarantee that the particular placement of the program on one computer's directory structure will be the same as another's.*

Second Argument—File Mode

The second argument of the *open* member function defines the *mode* in which the file should be opened. One choice is whether the file should be opened for writing, reading, or both. However, there are other choices, each called a file mode *flag*. Table 13-1 lists the file mode flags:

File Mode Flag	Description
ios::app	Append mode. The file's existing contents are preserved and all output is written to the end of the file.
ios::ate	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file. This flag usually is used with binary mode.
ios::binary	Binary mode. Information is written to the file in binary format, rather than in the default text format.
ios::in	Input mode. Information will be read from the file. The file will not be created if it does not exist.
ios::out	Output mode. Information will be written to the file. By default, the existing file contents will be overwritten.
ios::trunc	If the file already exists, its contents will be truncated, another word for deleted or overwritten. This is the default mode of ios::out.

Table 13-1 File Mode Flags

If you use the *ofstream* object to open a file, you do not *need* any file mode flags. Indeed, the examples in the previous section did not use any file mode flags. An *ofstream* object may only be used to open a file for writing, and cannot be used to open a file for reading. Therefore, there is no need to specify the *ios::out* flag; use of that flag is implied by use of the *ofstream* object to open the file.

However, you may want to use one or more file mode flags with the *open* member function of the *ofstream* object if you do not want the default, which is to open the file in text rather than binary mode and overwrite rather than append to the existing file contents. One example of when you might want to append is an error log file, which keeps track of errors that may occur in a program. When a new error occurs, you don't want to erase the history of prior errors, but rather you want to add to that history.

You can combine two or more flags when opening a file. For example, the following statements open a file in binary mode and to append rather than to overwrite. The two file mode flags are combined using the *bitwise or* operator (|):

```
ofstream outfile;
outfile.open("students.dat", ios::binary | ios::app);
```

NOTE: *The bitwise or operator | is not the same as the logical or operator || even though they share the name or and the keystroke |.*

While you don't need to specify any file mode flags if you use the *ofstream* object to open a file, you *should* specify file mode flags if you use the *fstream* object to open a file. Whereas an *ofstream* object may only be used to open a file for writing and not reading, an *fstream* object may be used for both purposes. Therefore, you should specify whether you are using the *open* member function of the *fstream* object to open the file for writing, reading, or both.

The following code fragment uses the *open* member function of the *fstream* object to open the file for writing only:

```
fstream afile;
afile.open("students.dat", ios::out);
```

The *fstream* or *ofstream* Constructor

You also may use the *fstream* or *ofstream* constructor to open a file for writing. A *constructor* is a function that is automatically called when you attempt to create an *instance* of an object.

An object instance is akin to a variable of a primitive data type, such as an int. For example, the following statement could be characterized as creating an instance, named *age,* of an integer:

```
int age;
```

Similarly, the following statement creates an *fstream* instance named *afile:*

```
fstream afile;
```

Object constructors may be overloaded, such that for the same object there may be a constructor with no arguments, a constructor with one argument, a constructor with two arguments, and so forth. For example, the previous statement, *fstream afile,* is called the no-argument constructor of the *fstream* object.

The following statement calls the one-argument constructor of the *ofstream* object, both creating an *ofstream* instance and opening the file *students.dat* for output:

```
ofstream outFile("students.dat", ios:out);
```

The following statement calls the two-argument constructor of the *fstream* object, both creating an *fstream* instance and opening the file *students.dat* for output:

```
fstream aFile("students.dat", ios:out);
```

In essence, declaring an *ofstream* (or *fstream*) variable in one statement and then calling the *open* member function in a second statement is analogous to declaring a primitive variable in one statement and then assigning it a value in a second statement, such as:

```
int age;
age = 39;
```

By contrast, using the one or two argument *ofstream* (or *fstream*) constructor is analogous to initializing a primitive variable, such as:

```
int age = 39;
```

One alternative is not inherently better than the other. Usually, the specific needs of a particular program will dictate which alternative better fits your needs.

Opening a File for Reading

The discussion in the previous section concerning opening a file for writing also applies to opening a file for reading. The primary difference is that the object that calls the *open* member function, or whose constructor you may use, may be, in addition to an *fstream* object, an *ifstream* object instead of an *ofstream* object. Additionally, the file to be opened for reading must already exist. Unlike opening a file for writing, attempting to open a file for reading will not automatically create it if it does not yet exist. This issue is discussed further in the next section.

The following statements use the *open* member function of the *ifstream* object to open a file for reading:

```
ifstream infile;
infile.open("students.dat");
```

You could accomplish the same purpose using the *fstream* object, specifying by a file mode flag that the file is being opened for reading only:

```
fstream afile;
afile.open("students.dat", ios::in);
```

The following statement uses the *ifstream* constructor to open a file for reading:

```
ifstream infile ("students.dat");
```

You could accomplish the same purpose using the *fstream* constructor, specifying in the second argument the file mode flag that the file is being opened for reading only:

```
fstream afile ("students.dat", ios::in);
```

Opening a File for Reading and Writing

You can use the *fstream* object to open a file for reading and for writing. You cannot use either the *ofstream* or *ifstream* object for this purpose, as an *ofstream* object cannot be used to read files, and an *ifstream* object cannot be used to write to files.

The following code fragment uses the *open* member function of the *fstream* object for this purpose:

```
fstream afile;
afile.open("students.dat", ios::in | ios::out);
```

Alternatively, you can use the two-argument *fstream* constructor:

```
fstream afile ("students.dat", ios::in | ios::out);
```

Both alternatives use the *bitwise or* operator (|) discussed in the earlier section "Second Argument—File Mode" to combine the file mode flags for input and output.

NOTE: Combining the ios::in and ios::out flags changes expected defaults. The ios::out flag by itself causes an existing file to be overwritten, and the ios::in flag by itself requires that the file already exist. However, when the ios::in and ios::out files are used together, the file's existing contents are preserved, and the file will be created if it does not already exist.

Checking if the File Was Opened

You should not assume that a file was successfully opened with the *open* member function or the constructor. There are several reasons why the file may not have been successfully opened. If the file was not successfully opened, but your code casually assumes it was and attempts to read from, or write to, the file, errors may occur.

The primary difference between opening a file for reading and for writing is that while you can write to a file that does not exist—the operating system simply creates the file—you cannot read from a file unless it already exists. Therefore, you should check if the file was opened successfully for reading before you attempt to read it.

If the file could not be opened for reading, then the value of the *ifstream* object that called the *open* function is NULL. As you may recall from Chapter 11, NULL is a constant defined in several standard library files whose value is zero.

Alternatively, if the file could not be opened for reading, then the *ifstream* object's *fail* member function returns true, which is the *fail* function's return value if a file operation, in this case attempting to open a file, was not successful.

The following code illustrates the use of both checking if the *ifstream* object used to call the *open* function is NULL and whether the *ifstream* object's *fail* member function returns true:

```
#include <fstream>
#include <iostream>
using namespace std;

int main ()
{
   ifstream infile;
   infile.open("students.dat");
   cout << "(infile) = " << infile << endl;
   cout << "(infile.fail()) = " << infile.fail() << endl;
   return 0;
}
```

If the *students.dat* file does not yet exist, the output would be

```
(infile) = 00000000
(infile.fail()) = 1
```

However, if there was a file named *students.dat* in the same directory as your program, then the output would be

```
(infile) = 0012FE40
(infile.fail()) = 0
```

The value, 0012FE40, is the address of the *ifstream* variable *infile,* and of course could be different if you run this program.

Unlike an *ifstream* object, an *ofstream* object that attempts to open a file that does not yet exist is not NULL, and its *fail* member function would return false, because the operating system will create the file if it does not already exist. However, opening a file for writing is not always successful. For example, before you run the following program, create a file named *students.dat* in the same directory as your program but, through its properties, check read only:

```
#include <fstream>
#include <iostream>
using namespace std;

int main ()
{
   ofstream outfile;
   outfile.open("students.dat");
   cout << "( outfile) = " << outfile << endl;
   cout << "( outfile.fail()) = " << outfile.fail() <<
endl;
   return 0;
}
```

The following output reflects that the *ofstream* object is NULL, and its *fail* function returns true, because you cannot open for writing a file that is read only.

```
(outfile) = 00000000
(outfile.fail()) = 1
```

If you cannot open a file for reading or writing, then you do not want to proceed to execute the code that reads from, or writes to, the file. Instead, you may want to stop execution of the function, as in the following code fragment:

```
ifstream infile;
infile.open("students.dat");
if (infile == NULL)
{
   cout << "Error in opening file for reading";
   return 0;
}
// code to read from file
```

NOTE: *For purposes of brevity and avoiding repetitive code, some of the following code in this chapter omits checking if a file was opened successfully.*

Closing a File

Of course, you are not going to close a file as soon as you open it. You will read or write to the file first. However, closing a file is relatively simple, so I will discuss this issue out of order before discussing the more complex subjects of writing to, and reading from, a file.

You should close a file when you are finished reading or writing to it. While the file object will be closed when the program ends, your program's performance will be improved if you close a file when you are finished with it because each open file requires system resources. Additionally, some operating systems limit the number of open "handles" to files. Finally, you will avoid a "sharing" problem caused by trying in one part of your program to open a file that in another part of the program previously was opened but not closed.

You close a file using, naturally enough, the *close* member function, which takes no arguments. The following example closes a file opened for writing:

```
ofstream outfile;
outfile.open("students.dat");
// do something
outfile.close();
```

The same syntax applies to closing a file for reading.

```
ifstream infile;
infile.open("students.dat");
// do something
infile.close();
```

Writing to a File

You output or write information to a file from your program using the *stream insertion* operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an *ofstream* or *fstream* object instead of the *cout* object.

The following program writes information inputted by the user to a file named *students.dat,* which is created if it does not already exist.

```
#include <fstream>
#include <iostream>
```

```
using namespace std;

int main ()
{
   char data[80];
   ofstream outfile;
   outfile.open("students.dat");
   cout << "Writing to the file" << endl;
   cout << "===================" << endl;
   cout << "Enter class name: ";
   cin.getline(data, 80);
   outfile << data << endl;
   cout << "Enter number of students: ";
   cin >> data;
   cin.ignore();
   outfile << data << endl;
   outfile.close();
   return 0;
}
```

The input and output could be

```
Writing to the file
===================
Enter class name: Programming Demystified
Enter number of students: 32
```

Open the file in a plain-text editor such as Notepad. The contents with the preceding sample input would be as follows:

```
Programming Demystified
32
```

The statement that wrote to the file included the *endl* keyword:

```
outfile << data << endl;
```

The reason is to write the name of the class ("Programming Demystified") to a different line than the number of students, 32. Otherwise, the file contents would be

```
Programming Demystified32
```

NOTE: The call to the ignore member function after cin >> data follows the advice in Chapter 12 to clear the newline character from the input buffer after using the cin object with the stream extraction operator (>>).

You instead could have used an *fstream* object to write to the file. You would have changed the data type of *outfile* from *ofstream* to *fstream* and then changed the call to the open method to include two arguments:

```
fstream outfile;
outfile.open("students.dat", ios::out);
```

Alternatively, you could have used the *fstream* constructor:

```
fstream outfile ("students.dat", ios::out);
```

If you want to append, you only need to add an *ios:app* flag to the second argument of the constructor or the *open* member function using the *bitwise or* operator (|).

Reading from a File

You input or read information from a file into your program using the *stream extraction* operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an *ifstream* (or *fstream*) object instead of the *cin* object.

The following program builds on the previous one. After writing information inputted by the user to a file named *students.dat,* the program reads information from the file and outputs it onto the screen:

```
#include <fstream>
#include <iostream>
using namespace std;

int main ()
{
   char data[80];
   ofstream outfile;
   outfile.open("students.dat");
   cout << "Writing to the file" << endl;
   cout << "===================" << endl;
   cout << "Enter class name: ";
   cin.getline(data, 80);
   outfile << data << endl;
   cout << "Enter number of students: ";
   cin >> data;
   cin.ignore();
   outfile << data << endl;
   outfile.close();
   ifstream infile;
   cout << "Reading from the file" << endl;
   cout << "=====================" << endl;
   infile.open("students.dat");
   infile >> data;
   cout << data << endl;
   infile >> data;
   cout << data << endl;
   infile.close();
   return 0;
}
```

Sample input and output:

```
Writing to the file
===================
Enter class name: Programming
Enter number of students: 32
Reading from the file
=====================
Programming
32
```

Reading a Line of a File

With the same program, try entering a class name with an embedded space. The following is some sample input and output:

```
Writing to the file
===================
Enter class name: Programming Demystified
Enter number of students: 32
Reading from the file
=====================
Programming
Demystified
```

The following are the contents of the file after the inputted data was written to it:

```
Programming Demystified
32
```

The first read of the file did not read the first line of the file, "Programming Demystified." Instead, the first read of the file only read the word "Programming" and then stopped. Consequently the second line of the program read the remainder of the first line of the file, "Demystified," instead of the number of students.

The *ifstream* object together with the *stream extraction* operator reads the file sequentially, starting with the first byte of the file. The first attempt to read the file starts at the beginning of the file and goes to the first whitespace character (a space, tab, or new line) or the end of the file, whichever comes first. The second attempt starts at the first printable character after that whitespace, and continues to the next whitespace character or the end of the file, whichever comes first.

The first read attempt only read "Programming," not "Programming Demystified," because the read stopped at the whitespace between "Programming" and "Demystified." The second attempt read "Demystified." There were no further read attempts, so the number of students, 32, was never read.

This should seem like déjà vu. We encountered a similar issue in Chapter 10 using the *cin* object with the *stream extraction* operator (>>). As in Chapter 10 with the *cin* object, the solution is to use *getline.*

If you are working with C-strings, then you should use the *getline* member function. The only difference between using the *getline* member function here and in Chapter 10 is that here the *getline* member function is called by an *ifstream* or *fstream* object instead of a *cin* object. Accordingly, we need to replace the two calls to *infile* >> *data* with the following:

```
infile.getline(data, 80);
```

You also can use *getline* with the C++ string class. The only difference between using the *getline* member function here and in Chapter 10 is that here the first argument of the *getline* member function is an *ifstream* or *fstream* object instead of a *cin* object. Accordingly, we need to replace the two calls to *infile* >> *data* with the following:

```
getline(infile, data);
```

The following modification of the previous program uses the *getline* function with the C++ string class:

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main ()
{
   string data;
   ofstream outfile;
   outfile.open("students.dat");
   cout << "Writing to the file" << endl;
   cout << "===================" << endl;
   cout << "Enter class name: ";
   getline(cin, data);
   outfile << data<< endl;
   cout << "Enter number of students: ";
   cin >> data;
   cin.ignore();
   outfile << data<< endl;
   outfile.close();
   ifstream infile;
   cout << "Reading from the file" << endl;
   cout << "=====================" << endl;
   infile.open("students.dat");
   getline(infile, data);
   cout << data << endl;
   getline(infile, data);
   cout << data << endl;
   infile.close();
   return 0;
}
```

As the following sample input and output reflects, the first read now reads the entire first line of the file even when that line contains embedded spaces:

```
Writing to the file
===================
Enter class name: Programming Demystified
Enter number of students: 32
Reading from the file
=====================
Programming Demystified
32
```

Looping Through the File

In the previous program, exactly two read attempts were made because we knew there were two lines of data in the file, no more, no less. However, often we may not know the number of pieces of data to be read. All we want is to read the file until we have reached the end of it.

The *ifstream* object has an *eof* function, *eof* being an abbreviation for end of file. This function, which takes no arguments, returns true if the end of the file has been reached, and false if otherwise.

However, the *eof* function is not as reliable with text files as it is with binary files in detecting the end of the file. The *eof* function's return value may not accurately reflect if the end of the file was reached if the last item in the file is followed by one or more whitespace characters. This is not an issue with binary files since they do not contain whitespace characters.

A better choice is the *fail* member function, discussed in the earlier section "Checking if the File Was Opened." The following code fragment shows how to use the *fail* member function in reading a file until the end of the file is reached:

```
ifstream infile;
infile.open("students.dat");
infile >> data;
while(!infile.fail())
{
   infile >> data;
   cout << data;
}
infile.close();
```

The preceding code fragment has two *infile >> data* statements, one before the loop begins, the other inside the loop. The reason is that the end of file is not detected until *after* a read attempt is made. Thus, if the *infile >> data* statement before the loop was omitted and the file was empty, the *cout << data* statement would execute before an attempt was made to detect if the end of file had been reached.

Note: A do while loop could be used instead of a while loop. This would dispense with the need to check for end of file before entering the loop, but add the requirement to check inside the loop if (using an if statement) end of file had been reached. This is the usual tradeoff between while and do while loops.

Modifying the previous program, the code now would read

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main ()
{
   string data;
   ofstream outfile;
   outfile.open("students.dat");
   cout << "Writing to the file" << endl;
   cout << "===================" << endl;
   cout << "Enter class name: ";
   getline(cin, data);
   outfile << data<< endl;
   cout << "Enter number of students: ";
   cin >> data;
   cin.ignore();
   outfile << data<< endl;
   outfile.close();
   ifstream infile;
   cout << "Reading from the file" << endl;
   cout << "=====================" << endl;
   infile.open("students.dat");
   getline(infile, data);
   while(!infile.fail())
   {
      cout << data << endl;
      getline(infile, data);
   }
   infile.close();
   return 0;
}
```

File Stream Objects as Function Arguments

Chapter 9 explained how you can use functions to make your code more modular. In that spirit, let's rewrite the previous program to add two functions, each to be called from *main: writeFile* to open a file for writing using an *ofstream* object, and *readFile* to open a file for reading using an *ifstream* object. Each function includes code to check if the file was opened successfully and returns a Boolean value indicating whether the file was opened successfully:

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
bool writeFile (ofstream&, char*);
bool readFile (ifstream&, char*);

int main ()
{
   string data;
   bool status;
   ofstream outfile;
   status = writeFile(outfile, "students.dat");
   if (!status)
   {
      cout << "File could not be opened for writing\n";
      cout << "Program terminating\n";
      return 0;
   }
   else
   {
      cout << "Writing to the file" << endl;
      cout << "===================" << endl;
      cout << "Enter class name: ";
      getline(cin, data);
      outfile << data<< endl;
      cout << "Enter number of students: ";
      cin >> data;
      cin.ignore();
      outfile << data<< endl;
      outfile.close();
   }
   ifstream infile;
   status = readFile(infile, "students.dat");
   if (!status)
   {
      cout << "File could not be opened for reading\n";
      cout << "Program terminating\n";
      return 0;
   }
   else
   {
      cout << "Reading from the file" << endl;
      cout << "=====================" << endl;
      getline(infile, data);
      while(!infile.fail())
      {
         cout << data << endl;
         getline(infile, data);
      }
      infile.close();
   }
   return 0;
}

bool writeFile (ofstream& file, char* strFile)
```

```
{
   file.open(strFile);
   if (file.fail())
         return false;
   else
         return true;
}

bool readFile (ifstream& ifile, char* strFile)
{
   ifile.open(strFile);
   if (ifile.fail())
         return false;
   else
         return true;
}
```

For each function, the file stream object is passed by reference instead of by value even though neither function changes the contents of the file. The reason is that the internal state of a file stream object may change with an open operation even if the contents of the file may not change.

Summary

Data is persistent when it survives after the program is finished or even after the computer is turned off. Data stored in variables does not persist because RAM, where the variables are stored, is cleared when the program (or the computer) stops running. It is necessary to save data to a file on the computer's hard drive or other storage medium so that data later can be retrieved when needed.

This chapter showed you how to make your data persistent by saving it to a file. Since saving the data accomplishes little unless you can later retrieve it, this chapter also showed you how to retrieve data from a file.

A file is a collection of data. It is located on persistent storage, such as a hard drive, CD-ROM, or other storage device.

Files store data in one of two formats, text and binary. Text files store data that has been converted into strings of ASCII characters. By contrast, binary files store data in the same format in which data is stored in RAM, fundamentally ones and zeroes. Notepad and other plain-text editors use text files. Binary files may store more complex data, and therefore are used in more complex programs, such as word processing, spreadsheet, or database programs.

You should include the *fstream* standard library when your program reads from, or writes to, files. This standard library defines three data types. The *ofstream* data type represents the output file stream, the direction of output being from your program out to a file. The *ifstream* data type represents the input file stream, the direction of input being from a file into your program. Finally, the *fstream* data type represents the file stream generally, and has the capabilities of both *ofstream* and *ifstream* in that it may both write information to files and read information from files.

The process of accessing a file, whether to read it, write to it, or both, goes through the following steps. First, the file first must be opened to establish a path of communication between the file and a stream object in your program—*fstream, ofstream,* or *ifstream*—used to access the file. Second, your program then reads from, or writes to, the file. Third, and finally, your program closes the file, using the *close* member function, to free system resources that are required to maintain the path of communication between the file and the stream object in your program, and also to avoid a "sharing" problem caused by trying in one part of your program to open a file that in another part of the program previously was opened but not closed.

You use either the *open* member function or a constructor to open a file. A constructor is a function that is automatically called when you attempt to create an *instance* of an object, such as an *fstream, ofstream,* or *ifstream* object. Either the *open* member function or a constructor may use two arguments. The first argument is the relative or absolute path to the file. The second argument, which may be optional, is one or more file mode flags, which define how the file should be opened, whether for input, output, appending, or something else.

You cannot assume that a file was successfully opened for reading or writing. You can use the *fail* member function to check if a file was successfully opened. You also can check to see if the file stream object used to open the file is NULL.

You write information to a file from your program using the *stream insertion* operator (<<) just as you use that operator to output information to the screen, except that you use an *ofstream* or *fstream* object instead of the *cout* object. Similarly, you read information from a file into your program using the *stream extraction* operator (>>) just as you use that operator to input information from the keyboard, except that you use an *ifstream* (or *fstream*) object instead of the *cin* object.

You read a line of a file using either the *getline* member function if you are working with C-strings or the *getline* function if you are working with the C++ string class. You use the *fail* member function to test for the end of the file as you read line by line through a file.

File stream objects may be passed as function arguments. They should be passed by reference rather than by value since the internal state of a file stream object may change with an open operation even if the contents of the file have not changed.

Quiz

1. What does it mean for data to be persistent?
2. What is a file?
3. What are the two formats in which files store data?
4. What standard library should you include when your program reads from, or writes to, files?
5. Which of the three objects, *fstream, ifstream,* or *ofstream,* may be used both for file input and file output?
6. What are the two functions you can use to open a file?
7. What is the purpose of opening a file?
8. What is the purpose of closing a file?
9. What is a constructor?
10. Which is a better choice for detecting end of file in a text file, the *eof* member function or the *fail* member function?
11. Should file stream objects be passed as function arguments by value or by reference?

The fast and [illegible] to understanding the fu[illegible]entals of C++

If you're looking for an easy way to learn C++ and want to immediately start writing your own programs, this is the resource you need. The hands-on approach and step-by-step instructions guide you through each phase of C++ programming with easy-to-understand language from start to finish.

Whether or not you have previous C++ experience, you'll get an excellent foundation here, discovering how computer programs and programming languages work. Next, you'll learn the basics of the language—what data types, variables, and operators are and what they do, then on to functions, arrays, loops, and beyond. With no unnecessary, time-consuming material included, plus quizzes at the end of each chapter and a final exam, you'll emerge a C++ pro, completing and running your very own complex programs in no time.

This one-of-a-kind self-teaching text offer

- An easy way to understand C++
- A quiz at the end of each chapter
- A final exam at the end of the book
- No unnecessary technical jargon
- A time-saving approach

Simple enough for a beginner, but challenging enough for an advanced student, *C++ Demystified* is your shortcut to mastering C++.

The McGraw-Hill Companies
McGraw-Hill Osborne
www.osborne.com
OSBORNE DELIVERS RESULTS!

PROGRAMMING/C++

7 83254 04349 7

ISBN 0-07-225370-3
51995
9 780072 253702

$19.95 USA | $28.95 CDN | £12.99 UK