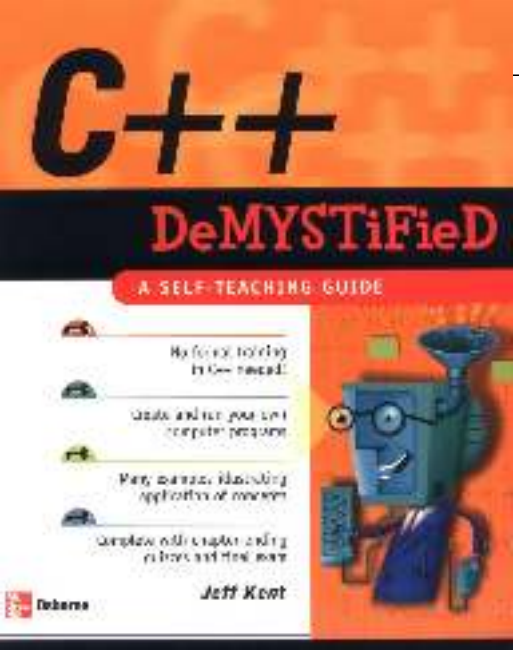
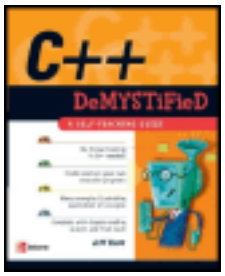

C++
DeMYSTiFieD
A SELF-TEACHING GUIDE
Jeff Kent
Osborne



C++ Demystified: A Self-Teaching Guide

by Jeff Kent

ISBN:0072253703

McGraw-Hill/Osborne © 2004

This hands-on, step-by-step resource will guide you through each phase of C++ programming, providing you with the foundation to discover how computer programs and programming languages work.

Table of Contents

C++ Demystified

Introduction

Chapter 1 - How a C++ Program Works

Chapter 2 - Memory and Data Types

Chapter 3 - Variables

Chapter 4 - Arithmetic Operators

Chapter 5 - Making Decisions: if and switch Statements

Chapter 6 - Nested if Statements and Logical Operators

Chapter 7 - The For Loop

Chapter 8 - While and Do While Loops

Chapter 9 - Functions

Chapter 10 - Arrays

Chapter 11 - What's the Address? Pointers

Chapter 12 - Character, C-String, and C++ String Class Functions

Chapter 13 - Persistent Data: File Input and Output

Chapter 14 - The Road Ahead: Structures and Classes

Final Exam

Answers to Quizzes and Final Exam

Index

List of Figures

List of Tables

Back Cover

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 instruction 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.

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.

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

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.

1234567890 FGR FGR 01987654

ISBN 0-07-225370-3

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.

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

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.

PREVIOUS NEXT

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

Overview

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).

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.

PREVIOUS NEXT

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.)

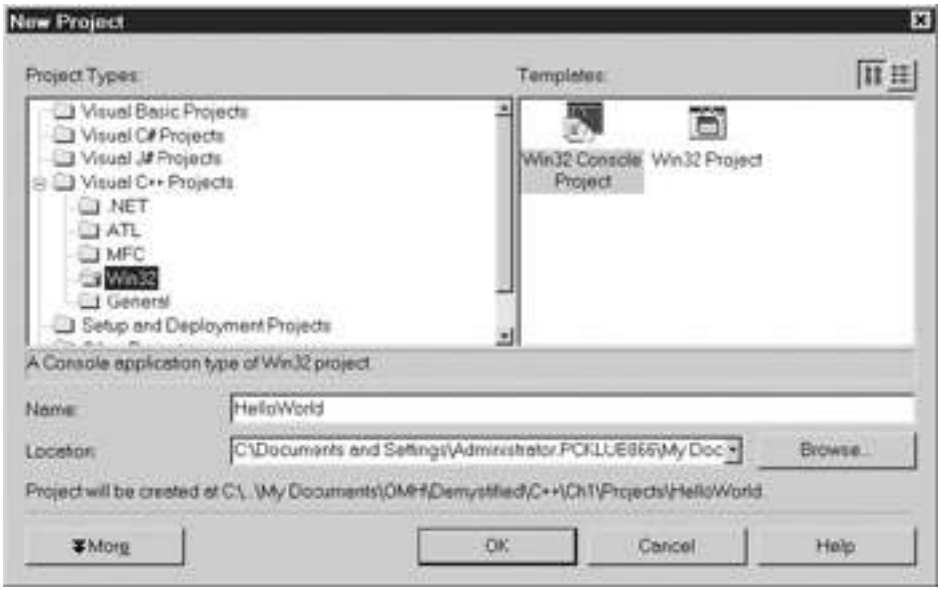


Figure 1-2: Creating a New Project

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.