

C++

By

EXAMPLE

Greg Perry

C++ By Example

© 1992 by Que

All rights reserved. Printed in the United States of America. No part of this book may be used or reproduced, in any form or by any means, or stored in a database or retrieval system, without prior written permission of the publisher except in the case of brief quotations embodied in critical articles and reviews. Making copies of any part of this book for any purpose other than your own personal use is a violation of United States copyright laws. For information, address Que, 11711 N. College Ave., Carmel, IN 46032.

Library of Congress Catalog Card Number: 92-64353

ISBN: 1-56529-038-0

This book is sold *as is*, without warranty of any kind, either express or implied, respecting the contents of this book, including but not limited to implied warranties for the book's quality, performance, merchantability, or fitness for any particular purpose. Neither Que Corporation nor its dealers or distributors shall be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this book.

96 95 94 93 92 8 7 6 5 4 3 2 1

Interpretation of the printing code: the rightmost double-digit number is the year of the book's printing; the rightmost single-digit number, the number of the book's printing. For example, a printing code of 92-1 shows that the first printing of the book occurred in 1992.

Publisher

Lloyd Short

Publishing Manager

Joseph Wikert

Development Editor

Stacy Hiquet

Production Editor

Kezia Endsley

Copy Editor

Bryan Gambrel

Technical Editor

Tim Moore

Editorial Assistants

Rosemarie Graham

Melissa Keegan

Book Design

Scott Cook

Michele Laseau

Production Analyst

Mary Beth Wakefield

Cover Design

Jean Bisesi

Indexer

Johnna VanHoose

Production

Caroline Roop (Book Shepherd)

Jeff Baker, Paula Carroll,

Michelle Cleary, Brook Farling,

Kate Godfrey, Bob LaRoche,

Laurie Lee, Jay Lesandrini,

Cindy L. Phipps, Linda Seifert,

Phil Worthington

Dedication

*Dr. Rick Burgess, you shaped my life. Good or bad, I'm what I am
thanks to your help. I appreciate the many hours we've shared together.
G.M.P.*

About the Author

Greg Perry has been a programmer and trainer for the past 14 years. He received his first degree in computer science, then he received a Masters degree in corporate finance. He currently is a professor of computer science at Tulsa Junior College, as well as a computer consultant and a lecturer. Greg Perry is the author of 11 other computer books, including *QBASIC By Example* and *C By Example*. In addition, he has published articles in several publications, including *PC World*, *Data Training*, and *Inside First Publisher*. He has attended computer conferences and trade shows in several countries, and is fluent in nine computer languages.

Acknowledgments

Much thanks to Stacy Hiquet and Joseph Wikert at Prentice Hall (Que) for trusting me completely with the direction and style of this book. The rest of my editors: Kezia Endsley, Bryan Gambrel, and the Technical Editor, Tim Moore, kept me on track so the readers can have an accurate and readable text.

The Tulsa Junior College administration continues to be supportive of my writing. More importantly, Diane Moore, head of our Business Services Division, Tony Hiram, and Elaine Harris, are friends who make teaching a joy and not a job.

As always, my beautiful bride Jayne, and my parents Glen and Bettye Perry, are my closest daily companions. It is for them I work.

Trademark Acknowledgments

Que Corporation has made every attempt to supply trademark information about company names, products, and services mentioned in this book. Trademarks indicated below were derived from various sources. Que Corporation cannot attest to the accuracy of this information.

AT&T is a registered trademark of American Telephone & Telegraph Company.

FORTRAN and COBOL are trademarks of International Business Machines Corporation (IBM).

Turbo BASIC is a registered trademark of Borland International, Inc.

Turbo C is a registered trademark of Borland International, Inc.

Microsoft QuickC and MS-DOS are registered trademarks of Microsoft Corporation.

ANSI is a registered trademark of American National Standards Institute.

Overview

I	Introduction to C++	
1	Welcome to C++	11
2	What Is a Program?	35
3	Your First C++ Program	51
4	Variables and Literals	69
5	Character Arrays and Strings	99
6	Preprocessor Directives	113
7	Simple Input/Output.....	133
II	Using C++ Operators	
8	Using C++ Math Operators and Precedence	163
9	Relational Operators	185
10	Logical Operators	207
11	Additional C++ Operators	221
III	C++ Constructs	
12	The while Loop	245
13	The for Loop	273
14	Other Loop Options	295
15	The switch and goto Statements	311
16	Writing C++ Functions	331
IV	Variable Scope and Modular Programming	
17	Variable Scope	353
18	Passing Values	379
19	Function Return Values and Prototypes	397
20	Default Arguments and Function Overloading	415
V	Character Input/Output and String Functions	
21	Device and Character Input/Output.....	431
22	Character, String, and Numeric Functions	449

VI	Arrays and Pointers	
23	Introducing Arrays	473
24	Array Processing	493
25	Multidimensional Arrays	519
26	Pointers	541
27	Pointers and Arrays	557
VII	Structures and File Input/Output	
28	Structures	583
29	Arrays of Structures	605
30	Sequential Files	625
31	Random-Access Files	645
32	Introduction to Object-Oriented Programming	661
VIII	References	
A	Memory Addressing, Binary, and Hexadecimal Review	679
B	Answers to Review Questions	701
C	ASCII Table	719
D	C++ Precedence Table	729
E	Keyword and Function Reference	733
F	The Mailing List Application	737
	Glossary	747
	Index	761

Contents

Introduction	1
Who Should Use This Book	1
The Book's Philosophy	2
Overview of This Book	2
Conventions Used in This Book	5
Index to the Icons	5
Margin Graphics (Book Diagrams)	6
Companion Disk Offer	8

I Introduction to C++

1 Welcome to C++	11
What C++ Can Do for You	12
The Background of C++	15
C++ Compared with Other Languages	16
C++ and Microcomputers	17
An Overview of Your Computer	19
Hardware	19
Software	29
Review Questions	33
Summary	34
2 What Is a Program?	35
Computer Programs	36
Program Design	38
Using a Program Editor	40
Using a C++ Compiler	42
Running a Sample Program	44
Handling Errors	46
Review Questions	48
Summary	49

3	Your First C++ Program	51
	Looking at a C++ Program.....	52
	The Format of a C++ Program	53
	Readability Is the Key	54
	Uppercase Versus Lowercase	55
	Braces and main()	56
	Comments in C++	57
	Explaining the Sample Program	60
	Review Questions	66
	Summary	67
4	Variables and Literals	69
	Variables	70
	Naming Variables	70
	Variable Types	72
	Declaring Variables	73
	Looking at Data Types	75
	Assigning Values to Variables	80
	Literals.....	82
	Assigning Integer Literals.....	83
	Assigning String Literals.....	85
	Assigning Character Literals	89
	Constant Variables	94
	Review Questions	95
	Review Exercises	97
	Summary	97
5	Character Arrays and Strings	99
	Character Arrays	100
	Character Arrays Versus Strings	103
	Review Questions	110
	Review Exercises	111
	Summary	111
6	Preprocessor Directives	113
	Understanding Preprocessor Directives	114
	The #include Directive	115
	The #define Directive	120

	Review Questions	128
	Review Exercises	130
	Summary	130
7	Simple Input/Output	133
	The cout Operator	134
	Printing Strings	134
	The cin Operator	144
	printf() and scanf()	149
	The printf() Function	149
	Conversion Characters	151
	The scanf() Function	154
	Review Questions	157
	Review Exercises	158
	Summary	159
II Using C++ Operators		
8	Using C++ Math Operators and Precedence	163
	C++'s Primary Math Operators	164
	The Unary Operators	165
	Division and Modulus	167
	The Order of Precedence	168
	Using Parentheses	170
	The Assignment Statements	174
	Multiple Assignments	175
	Compound Assignments	176
	Mixing Data Types in Calculations	178
	Type Casting	179
	Review Questions	182
	Review Exercises	183
	Summary	184
9	Relational Operators	185
	Defining Relational Operators	186
	The if Statement	189
	The else Statement	199

	Review Questions	203
	Review Exercises	204
	Summary	205
10	Logical Operators	207
	Defining Logical Operators.....	207
	Logical Operators and Their Uses	209
	C++'s Logical Efficiency.....	211
	Logical Operators and Their Precedence	216
	Review Questions	217
	Review Exercises	218
	Summary	219
11	Additional C++ Operators	221
	The Conditional Operator	222
	The Increment and Decrement Operators	225
	The sizeof Operator	230
	The Comma Operator	232
	Bitwise Operators	234
	Bitwise Logical Operators.....	235
	Review Questions	242
	Review Exercises	243
	Summary	243
III C++ Constructs		
12	The while Loop	245
	The while Statement.....	246
	The Concept of Loops	247
	The do-while Loop	252
	The if Loop Versus the while Loop.....	255
	The exit() Function and break Statement	256
	Counters and Totals	260
	Producing Totals	265
	Review Questions	268
	Review Exercises	269
	Summary	270

13	The for Loop	273
	The for Statement	274
	The Concept of for Loops	274
	Nested for Loops	286
	Review Questions	292
	Review Exercises	293
	Summary	293
14	Other Loop Options	295
	Timing Loops	296
	The break and for Statements	298
	The continue Statement	303
	Review Questions	308
	Review Exercises	308
	Summary	309
15	The switch and goto Statements	311
	The switch Statement	312
	The goto Statement	321
	Review Questions	327
	Review Exercises	328
	Summary	328
16	Writing C++ Functions	331
	Function Basics	332
	Breaking Down Problems	333
	More Function Basics	335
	Calling and Returning Functions	337
	Review Questions	349
	Summary	350
IV Variable Scope and Modular Programming		
17	Variable Scope	353
	Global Versus Local Variables	354
	Defining Variable Scope	355
	Use Global Variables Sparingly	362
	The Need for Passing Variables	363
	Automatic Versus Static Variables	369

	Three Issues of Parameter Passing	374
	Review Questions	375
	Review Exercises	375
	Summary	377
18	Passing Values	379
	Passing by Value (by Copy)	379
	Passing by Address (by Reference)	385
	Variable Addresses	385
	Sample Program	386
	Passing Nonarrays by Address	391
	Review Questions	394
	Review Exercises	395
	Summary	396
19	Function Return Values and Prototypes ...	397
	Function Return Values	398
	Function Prototypes	405
	Prototype for Safety	407
	Prototype All Functions	407
	Review Questions	412
	Review Exercises	412
	Summary	413
20	Default Arguments and Function Overloading	415
	Default Argument Lists	416
	Multiple Default Arguments	417
	Overloaded Functions	420
	Review Questions	426
	Review Exercises	426
	Summary	427
V	Character Input/Output and String Functions	
21	Device and Character Input/Output	431
	Stream and Character I/O	432
	Standard Devices	434
	Redirecting Devices from MS-DOS	435

	Printing Formatted Output to the Printer	436
	Character I/O Functions	437
	The get() and put() Functions	438
	The getch() and putch() Functions	444
	Review Questions	446
	Review Exercises	447
	Summary	448
22	Character, String, and Numeric Functions	449
	Character Functions	450
	Character Testing Functions	450
	Alphabetic and Digital Testing	450
	Special Character-Testing Functions	453
	Character Conversion Functions	453
	String Functions	455
	Useful String Functions	456
	String I/O Functions	456
	Converting Strings to Numbers	460
	Numeric Functions	461
	Useful Mathematical Functions	461
	Trigonometric Functions	464
	Logarithmic Functions	465
	Random-Number Processing	465
	Review Questions	467
	Review Exercises	468
	Summary	469
 VI Arrays and Pointers		
23	Introducing Arrays	473
	Array Basics	474
	Initializing Arrays	479
	Initializing Elements at Declaration Time	479
	Initializing Elements in the Program	486
	Review Questions	491
	Review Exercises	491
	Summary	492

24	Array Processing	493
	Searching Arrays	494
	Searching for Values	496
	Sorting Arrays	501
	Advanced Referencing of Arrays	508
	Review Questions	515
	Review Exercises	516
	Summary	517
25	Multidimensional Arrays	519
	Multidimensional Array Basics	520
	Reserving Multidimensional Arrays	522
	Mapping Arrays to Memory	524
	Defining Multidimensional Arrays	526
	Tables and for Loops	530
	Review Questions	537
	Review Exercises	538
	Summary	538
26	Pointers	541
	Introduction to Pointer Variables	542
	Declaring Pointers	543
	Assigning Values to Pointers	545
	Pointers and Parameters	546
	Arrays of Pointers	551
	Review Questions	553
	Summary	555
27	Pointers and Arrays	557
	Array Names as Pointers	558
	Pointer Advantages	560
	Using Character Pointers	563
	Pointer Arithmetic	568
	Arrays of Strings	574
	Review Questions	578
	Review Exercises	579
	Summary	580

VII Structures and File Input/Output

28	Structures	583
	Introduction to Structures	584
	Defining Structures	587
	Initializing Structure Data	591
	Nested Structures	600
	Review Questions	603
	Review Exercises	604
	Summary	604
29	Arrays of Structures	605
	Declaring Arrays of Structures	606
	Arrays as Members	615
	Review Questions	623
	Review Exercises	624
	Summary	624
30	Sequential Files	625
	Why Use a Disk?	626
	Types of Disk File Access	627
	Sequential File Concepts	628
	Opening and Closing Files	629
	Writing to a File	635
	Writing to a Printer	637
	Adding to a File	638
	Reading from a File	639
	Review Questions	642
	Review Exercises	643
	Summary	644
31	Random-Access Files	645
	Random File Records	646
	Opening Random-Access Files	647
	The seekg() Function	649
	Other Helpful I/O Functions	656
	Review Questions	658
	Review Exercises	658
	Summary	659

32	Introduction to Object-Oriented Programming	661
	What Is a Class?	662
	Data Members	662
	Member Functions	662
	Default Member Arguments	670
	Class Member Visibility	674
	Review Questions	676
	Review Exercise	676
	Summary	676

VIII References

A	Memory Addressing, Binary, and Hexadecimal Review	679
	Computer Memory	680
	Memory and Disk Measurements	680
	Memory Addresses	681
	Bits and Bytes	682
	The Order of Bits	686
	Binary Numbers	686
	Binary Arithmetic	690
	Binary Negative Numbers	692
	Hexadecimal Numbers	695
	Why Learn Hexadecimal?	697
	How Binary and Addressing Relate to C++	698
B	Answers to Review Questions	701
C	ASCII Table	719
D	C++ Precedence Table	729
E	Keyword and Function Reference	733
	stdio.h	734
	ctype.h	734
	string.h	735
	math.h	735
	stdlib.h	735

EXAMPLE

F The Mailing List Application	737
Glossary	747
Index	761

Introduction

Every day, more and more people learn and use the C++ programming language. I have taught C to thousands of students in my life. I see many of those students now moving to C++ in their school work or career. The C++ language is becoming an industry-accepted standard programming language, using the solid foundation of C to gain a foothold. C++ is simply a better C than C.

C++ By Example is one of several books in Que's new line of *By Example* series. The philosophy of these books is simple: The best way to teach computer programming concepts is with multiple examples. Command descriptions, format syntax, and language references are not enough to teach a newcomer a programming language. Only by looking at numerous examples and by running sample programs can programming students get more than just a "feel" for the language.

Who Should Use This Book

This book teaches at three levels: beginning, intermediate, and advanced. Text and numerous examples are aimed at each level. If you are new to C++, and even if you are new to computers, this book attempts to put you at ease and gradually build your C++ programming skills. If you are an expert at C++, this book provides a few extras for you along the way.

The Book's Philosophy

This book focuses on programming *correctly* in C++ by teaching structured programming techniques and proper program design. Emphasis is always placed on a program's readability rather than “tricks of the trade” code examples. In this changing world, programs should be clear, properly structured, and well-documented, and this book does not waver from the importance of this philosophy.

This book teaches you C++ using a holistic approach. In addition to learning the mechanics of the language, you learn tips and warnings, how to use C++ for different types of applications, and a little of the history and interesting asides about the computing industry.

Many other books build single applications, adding to them a little at a time with each chapter. The chapters of this book are stand-alone chapters, and show you complete programs that fully demonstrate the commands discussed in the chapter. There is a program for every level of reader, from beginning to advanced.

This book contains almost 200 sample program listings. These programs show ways that you can use C++ for personal finance, school and business record keeping, math and science, and general-purpose applications that almost everybody with a computer can use. This wide variety of programs show you that C++ is a very powerful language that is easy to learn and use.

Appendix F, “The Mailing List Application,” is a complete application—much longer than any of the other programs in the book—that brings together your entire working knowledge of C++. The application is a computerized mailing-list manager. Throughout the chapters that come before the program, you learn how each command in the program works. You can modify the program to better suit your own needs. (The comments in the program suggest changes you can make.)

Overview of This Book

This book is divided into eight parts. Part I introduces you to the C++ environment, as well as introductory programming concepts. Starting with Part II, the book presents the C++ programming

language commands and built-in functions. After mastering the language, you can use the book as a handy reference. When you need help with a specific C++ programming problem, turn to the appropriate area that describes that part of the language to see numerous examples of code.

To get an idea of the book's layout, read the following description of each section of the book:

Part I: Introduction to C++

This section explains what C++ is by describing a brief history of the C++ programming language and presenting an overview of C++'s advantages over other languages. This part describes your computer's hardware, how you develop your C++ programs, and the steps you follow to enter and run programs. You begin to write C++ programs in Chapter 3.

Part II: Using C++ Operators

This section teaches the entire set of C++ operators. The rich assortment of operators (more than any other programming language except APL) makes up for the fact that the C++ programming language is very small. The operators and their order of precedence are more important to C++ than most programming languages.

Part III: C++ Constructs

C++ data processing is most powerful due to the looping, comparison, and selection constructs that C++ offers. This part shows you how to write programs flowing with control computations that produce accurate and readable code.

Part IV: Variable Scope and Modular Programming

To support true structured programming techniques, C++ must allow for local and global variables, as well as offer several

ways to pass and return variables between functions. C++ is a very strong structured language that attempts, if the programmer is willing to “listen to the language,” to protect local variables by making them visible only to the parts of the program that need them.

Part V: Character Input/Output and String Functions

C++ contains no commands that perform input or output. To make up for this apparent oversight, C++ compiler writers supply several useful input and output functions. By separating input and output functions from the language, C++ achieves better portability between computers; if your program runs on one computer, it will work on any other.

This part also describes several of the other built-in math, character, and string functions available with C++. These functions keep you from having to write your own routines to perform common tasks.

Part VI: Arrays and Pointers

C++ offers single and multidimensional arrays that hold multiple occurrences of repeating data, but that do not require much effort on your part to process.

Unlike many other programming languages, C++ also uses pointer variables a great deal. Pointer variables and arrays work together to give you flexible data storage that allow for easy sorting and searching of data.

Part VII: Structures and File Input/Output

Variables, arrays, and pointers are not enough to hold the types of data that your programs require. Structures allow for more powerful grouping of many different kinds of data into manageable units.

Your computer would be too limiting if you could not store data to the disk and retrieve that data back in your programs. Disk

files are required by most “real world” applications. This section describes how C++ processes sequential and random-access files and teaches the fundamental principles needed to effectively save data to the disk. The last chapter in this section introduces object-oriented programming and its use of *classes*.

Part VIII: References

This final section of the book includes a reference guide to the ASCII table, the C++ precedence table, and to keywords and functions in C++. Also in this section are the mailing list application and the answers to the review questions.

Conventions Used in This Book

The following typographic conventions are used in this book:

- ◆ Code lines, variables, and any text you see on-screen are in monospace.
- ◆ Placeholders on format lines are in *italic monospace*.
- ◆ Filenames are in regular text, all uppercase (CCDOUB.CPP).
- ◆ Optional parameters on format lines are enclosed in flat brackets ([]). You do **not** type the brackets when you include these parameters.
- ◆ New terms, which are also found in the glossary, are in *italic*.

Index to the Icons

The following icons appear throughout this book:



Level 1 difficulty



Level 2 difficulty



Level 3 difficulty



Tip



Note



Caution



Pseudocode

The pseudocode icon appears beside pseudocode, which is typeset in *italic* immediately before the program. The pseudocode consists of one or more sentences indicating what the program instructions are doing, in English. Pseudocode appears before selected programs.









Margin Graphics (Book Diagrams)

To help your understanding of C++ further, this book includes numerous *margin graphics*. These margin graphics are similar to *flowcharts* you have seen before. Both use standard symbols to represent program logic. If you have heard of the adage “A picture is worth a thousand words,” you will understand why it is easier to look at the margin graphics and grasp the overall logic before dissecting programs line-by-line.

EXAMPLE

Throughout this book, these margin graphics are used in two places. Some graphics appear when a new command is introduced, to explain how the command operates. Others appear when new commands appear in sample programs for the first time.

The margin graphics do not provide complete, detailed explanations of every statement in each program. They are simple instructions and provide an overview of the new statements in question. The symbols used in the margin graphics, along with descriptions of them, follow:

	Terminal symbol ({, }, Return...)
	Assignment statement (total = total + newvalue; ctr = ctr + 1; ...)
	Input/output (scanf, printf...)
	Calling a function
	Small circle; loop begin
	Large dot; beginning and end of IF-THEN, IF-THEN-ELSE, and Switch
	Input/output of arrays; assumes implied FOR loop(s) needed to deal with array I/O
	Comment bracket; used for added info, such as name of a function

The margin graphics, the program listings, the program comments, and the program descriptions in the book provide many vehicles for learning the C++ language!

Companion Disk Offer

If you'd like to save yourself hours of tedious typing, use the order form in the back of this book to order the companion disk for *C++ By Example*. This disk contains the source code for all complete programs and sample code in this book, as well as the mailing-list application that appears in Appendix F. Additionally, the answers to many of the review exercises are included on the disk.

Part I

Introduction to C++

Welcome to C++

C++ is a recent addition to the long list of programming languages now available. Experts predict that C++ will become one of the most widely used programming languages within two to three years. Scan your local computer bookstore's shelves and you will see that C++ is taking the programming world by storm. More and more companies are offering C++ compilers. In the world of PCs, both Borland and Microsoft, two of the leading names of PC software, offer full-featured C++ compilers.

Although the C++ language is fairly new, having become popular within the last three years, the designers of C++ compilers are perfecting this efficient, standardized language that should soon be compatible with almost every computer in the world. Whether you are a beginning, an intermediate, or an expert programmer, C++ has the programming tools you need to make your computer do just what *you* want it to do. This chapter introduces you to C++, briefly describes its history, compares C++ to its predecessor C, shows you the advantages of C++, and concludes by introducing you to hardware and software concepts.

What C++ Can Do for You

C++ is currently defined by American Telephone & Telegraph, Incorporated, to achieve conformity between versions of C++.

Imagine a language that makes your computer perform to your personal specifications! Maybe you have looked for a program that keeps track of your household budget—exactly as you prefer—but haven't found one. Perhaps you want to track the records of a small (or large) business with your computer, but you haven't found a program that prints reports exactly as you'd like them. Possibly you have thought of a new and innovative use for a computer and you would like to implement your idea. C++ gives you the power to develop all these uses for your computer.

If your computer could understand English, you would not have to learn a programming language. But because it does not understand English, you must learn to write instructions in a language your computer recognizes. C++ is a powerful programming language. Several companies have written different versions of C++, but almost all C++ languages available today conform to the AT&T standard. *AT&T-compatible* means the C++ language in question conforms to the standard defined by the company that invented the language, namely, American Telephone & Telegraph, Incorporated. AT&T realizes that C++ is still new and has not fully matured. The good people there just completed the AT&T C++ 3.0 standard to which software companies can conform. By developing a uniform C++ language, AT&T helps ensure that programs you write today will most likely be compatible with the C++ compilers of tomorrow.



NOTE: The AT&T C++ standard is only a suggestion. Software companies do not have to follow the AT&T standard, although most choose to do so. No typical computer standards committee has yet adopted a C++ standard language. The committees are currently working on the issue, but they are probably waiting for C++ to entrench the programming community before settling on a standard.

C++ is called a "better C than C."

Companies do not have to follow the AT&T C++ 3.0 standard. Many do, but add their own extensions and create their own version to do more work than the AT&T standard includes. If you are using the AT&T C++ standard, your program should successfully run on any other computer that also uses AT&T C++.

AT&T developed C++ as an improved version of the C programming language. C has been around since the 1970s and has matured into a solid, extremely popular programming language. ANSI, the American National Standards Institute, established a standard C programming specification called ANSI C. If your C compiler conforms to ANSI C, your program will work on any other computer that also has ANSI C. This compatibility between computers is so important that AT&T's C++ 3.0 standard includes almost every element of the ANSI C, plus more. In fact, the ANSI C committee often requires that a C++ feature be included in subsequent versions of C. For instance, function prototypes, a feature not found in older versions of ANSI C, is now a requirement for approval by the ANSI committee. Function prototypes did not exist until AT&T required them in their early C++ specification.

C++ By Example teaches you to program in C++. All programs conform to the AT&T C++ 2.1 standard. The differences between AT&T 2.1 and 3.0 are relatively minor for beginning programmers. As you progress in your programming skills, you will want to tackle the more advanced aspects of C++ and Version 3.0 will come more into play later. Whether you use a PC, a minicomputer, a mainframe, or a supercomputer, the C++ language you learn here should work on any that conform to AT&T C++ 2.1 and later.

There is a debate in the programming community as to whether a person should learn C before C++ or learn only C++. Because C++ is termed a "better C," many feel that C++ is an important language in its own right and can be learned just as easily as C. Actually, C++ pundits state that C++ teaches better programming habits than the plain, "vanilla" C. This book is aimed at the beginner programmer, and the author feels that C++ is a great language with which to begin. If you were to first learn C, you would have to "unlearn" a few things when you moved to C++. This book attempts to use the C++ language elements that are better than C. If you are new to programming, you learn C++ from the start. If you have a C background, you learn that C++ overcomes many of C's limitations.

When some people attempt to learn C++ (and C), even if they are programmers in other computer languages, they find that C++ can be cryptic and difficult to understand. This does not have to be the case. When taught to write clear and concise C++ code in an order that builds on fundamental programming concepts,

programmers find that C++ is no more difficult to learn or use than any other programming language. Actually, after you start using it, C++'s modularity makes it even easier to use than most other languages. Once you master the programming elements this book teaches you, you will be ready for the advanced power for which C++ was designed—*object-oriented programming* (OOP). The last chapter of this book, “Introduction to Object-Oriented Programming,” offers you the springboard to move to this exciting way of writing programs.

Even if you've never programmed a computer before, you will soon understand that programming in C++ is rewarding. Becoming an expert programmer in C++—or in any other computer language—takes time and dedication. Nevertheless, you can start writing simple programs with little effort. After you learn the fundamentals of C++ programming, you can build on what you learn and hone your skills as you write more powerful programs. You also might see new uses for your computer and develop programs others can use.

The importance of C++ cannot be overemphasized. Over the years, several programming languages were designed to be “the only programming language you would ever need.” PL/I was heralded as such in the early 1960s. It turned out to be so large and took so many system resources that it simply became another language programmers used, along with COBOL, FORTRAN, and many others. In the mid-1970s, Pascal was developed for smaller computers. Microcomputers had just been invented, and the Pascal language was small enough to fit in their limited memory space while still offering advantages over many other languages. Pascal became popular and is still used often today, but it never became *the* answer for all programming tasks, and it failed at being “the only programming language you would ever need.”

When the mass computer markets became familiar with C in the late 1970s, C also was promoted as “the only programming language you would ever need.” What has surprised so many skeptics (including this author) is that C has practically fulfilled this promise! An incredible number of programming shops have converted to C. The appeal of C's efficiency, combined with its portability among computers, makes it the language of choice. Most of

today's familiar spreadsheets, databases, and word processors are written in C. Now that C++ has improved on C, programmers are retooling their minds to think in C++ as well.

The programmer help-wanted ads seek more and more C++ programmers every day. By learning this popular language, you will be learning the latest direction of programming and keeping your skills current with the market. You have taken the first step: with this book, you learn the C++ language particulars as well as many programming tips to use and pitfalls to avoid. This book attempts to teach you to be not just a C++ programmer, but a better programmer by applying the structured, long-term programming habits that professionals require in today's business and industry.

The Background of C++

The UNIX operating system was written almost entirely in C.

Before you jump into C++, you might find it helpful to know a little about the evolution of the C++ programming language. C++ is so deeply rooted in C that you should first see where C began. Bell Labs first developed the C programming language in the early 1970s, primarily so Bell programmers could write their UNIX operating system for a new DEC (Digital Equipment Corporation) computer. Until that time, operating systems were written in assembly language, which is tedious, time-consuming, and difficult to maintain. The Bell Labs people knew they needed a higher-level programming language to implement their project quicker and create code that was easier to maintain.

Because other high-level languages at the time (COBOL, FORTRAN, PL/I, and Algol) were too slow for an operating system's code, the Bell Labs programmers decided to write their own language. They based their new language on Algol and BCPL. Algol is still used in the European markets, but is not used much in America. BCPL strongly influenced C, although it did not offer the various data types that the makers of C required. After a few versions, these Bell programmers developed a language that met their goals well. C is efficient (it is sometimes called a high, low-level language due to its speed of execution), flexible, and contains the proper language elements that enable it to be maintained over time.

In the 1980s, Bjourn Stroustrup, working for AT&T, took the C language to its next progression. Mr. Stroustrup added features to compensate for some of the pitfalls C allowed and changed the way programmers view programs by adding object-orientation to the language. The object-orientation aspect of programming started in other languages, such as Smalltalk. Mr. Stroustrup realized that C++ programmers needed the flexibility and modularity offered by a true OOP programming language.

C++ Compared with Other Languages

C++ requires more stringent data-type checking than does C.

If you have programmed before, you should understand a little about how C++ differs from other programming languages on the market. C++ is efficient and has much stronger typing than its C predecessor. C is known as a *weakly typed* language; variable data types do not necessarily have to hold the same type of data. (Function prototyping and type casting help to alleviate this problem.)

For example, if you declare an integer variable and decide to put a character value in it, C enables you to do so. The data might not be in the format you expect, but C does its best. This is much different from stronger-typed languages such as COBOL and Pascal.

If this discussion seems a little over your head at this point, relax. The upcoming chapters will elaborate on these topics and provide many examples.

C++ is a small, block-structured programming language. It has fewer than 46 keywords. To compensate for its small vocabulary, C++ has one of the largest assortment of *operators* such as +, -, and && (second only to APL). The large number of operators in C++ might tempt programmers to write cryptic programs that have only a small amount of code. As you learn throughout this book, however, you will find that making the program more readable is more important than saving some bytes. This book teaches you how to use the C++ operators to their fullest extent, while maintaining readable programs.

C++'s large number of operators (almost equal to the number of keywords) requires a more judicious use of an *operator precedence*

table. Appendix D, “C++ Precedence Table,” includes the C++ operator precedence table. Unlike most other languages that have only four or five levels of precedence, C++ has 15. As you learn C++, you have to master each of these 15 levels. This is not as difficult as it sounds, but its importance cannot be overstated.

C++ also has no input or output statements. You might want to read that sentence again! C++ has no commands that perform input or output. This is one of the most important reasons why C++ is available on so many different computers. The I/O (input/output) statements of most languages tie those languages to specific hardware. BASIC, for instance, has almost twenty I/O commands—some of which write to the screen, to the printer, to a modem, and so forth. If you write a BASIC program for a microcomputer, chances are good that it cannot run on a mainframe without considerable modification.

C++’s input and output are performed through the abundant use of operators and function calls. With every C++ compiler comes a library of standard I/O functions. I/O functions are *hardware independent*, because they work on any device and on any computer that conform to the AT&T C++ standard.

To master C++ completely, you have to be more aware of your computer’s hardware than most other languages would require you to be. You certainly do not have to be a hardware expert, but understanding the internal data representation makes C++ much more usable and meaningful.

It also helps if you can become familiar with binary and hexadecimal numbers. You might want to read Appendix A, “Memory Addressing, Binary, and Hexadecimal Review,” for a tutorial on these topics before you start to learn the C++ language. If you do not want to learn these topics, you can still become a good C++ programmer, but knowing what goes on “under the hood” makes C++ more meaningful to you as you learn it.

C++ and Microcomputers

C was a relatively unknown language until it was placed on the microcomputer. With the invention and growth of the microcomputer, C blossomed into a worldwide computer language. C++

extends that use on smaller computers. Most of readers of *C++ By Example* are probably working on a microcomputer-based C++ system. If you are new to computers, this section will help you learn how microcomputers were developed.

In the 1970s, NASA created the *microchip*, a tiny wafer of silicon that occupies a space smaller than a postage stamp. Computer components were placed on these microchips, hence computers required much less space than before. NASA produced these smaller computers in response to their need to send rocket ships to the moon with on-board computers. The computers on Earth could not provide split-second accuracy for rockets because radio waves took several seconds to travel between the Earth and the moon. Through development, these microchips became small enough so the computers could travel with a rocket and safely compute the rocket's trajectory.

The space program was not the only beneficiary of computer miniaturization. Because microchips became the heart of the *microcomputer*, computers could now fit on desktops. These microcomputers cost much less than their larger counterparts, so many people started buying them. Thus, the home and small-business computer market was born.

Today, microcomputers are typically called *PCs* from the widespread use of the original IBM PC. The early PCs did not have the memory capacity of the large computers used by government and big business. Nevertheless, PC owners still needed a way to program these machines. BASIC was the first programming language used on PCs. Over the years, many other languages were ported from larger computers to the PC. However, no language was as successful as C in becoming the worldwide standard programming language. C++ seems to be the next standard.

Before diving into C++, you might take a few moments to familiarize yourself with some of the hardware and software components of your PC. The next section, "An Overview of Your Computer," introduces you to computer components that C++ interacts with, such as the operating system, memory, disks, and I/O devices. If you are already familiar with your computer's hardware and software, you might want to skip to Chapter 2, "What Is a Program?," and begin using C++.

An Overview of Your Computer

Your computer system consists of two parts: *hardware* and *software*. The hardware consists of all the physical parts of the machine. Hardware has been defined as “anything you can kick.” Although this definition is coarse, it illustrates that your computer’s hardware consists of the physical components of your PC. The software is everything else. Software comprises the programs and data that interact with your hardware. The C++ language is an example of software. You can use C++ to create even more software programs and data.

Hardware

Figure 1.1 shows you a typical PC system. Before using C++, you should have a general understanding of what hardware is and how your hardware components work together.

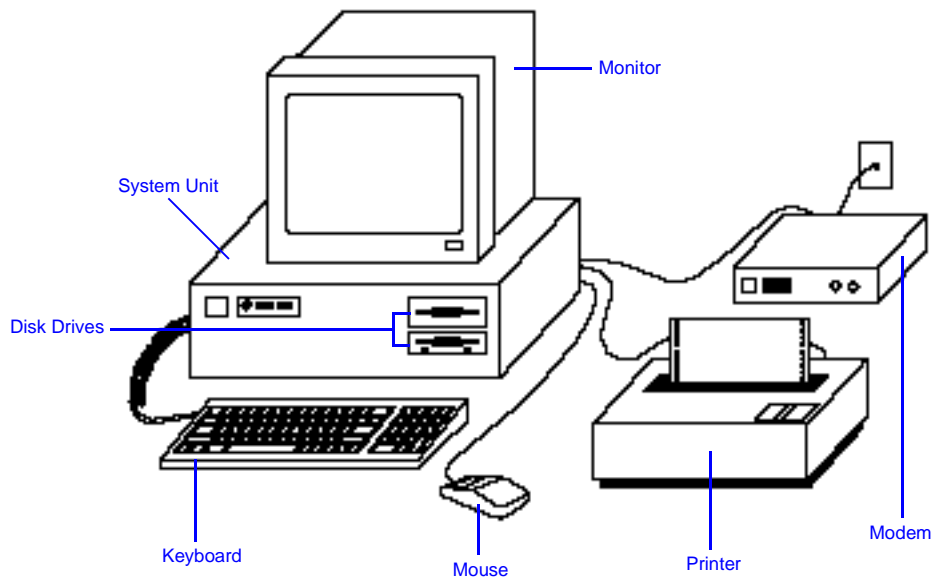


Figure 1.1. A typical PC system.

The System Unit and Memory

The *system unit* is the large, box-shaped component of the computer. This unit houses the PC's microprocessor. You might hear the microprocessor called the *CPU*, or *central processing unit*. The CPU acts like a traffic cop, directing the flow of information throughout your computer system. The CPU is analogous also to the human brain. When you use a computer, you are actually interacting with its CPU. All the other hardware exists so the CPU can send information to you (through the monitor or the printer), and you can give instructions to the CPU (through the keyboard or the mouse).

The CPU also houses the computer's internal *memory*. Although the memory has several names, it is commonly referred to as *RAM* (random-access memory). RAM is where the CPU looks for software and data. When you run a C++ program, for example, you are instructing your computer's CPU to look in RAM for that program and carry out its instructions. C++ uses RAM space when it is loaded.

A byte is a single character of memory.

RAM is used for many things and is one of the most important components of your computer's hardware. Without RAM, your computer would have no place for its instructions and data. The amount of RAM can also affect the computer's speed. In general, the more RAM your computer has, the more work it can do and the faster it can process data.

The amount of RAM is measured by the number of characters it can hold. PCs generally hold approximately 640,000 characters of RAM. A character in computer terminology is called a *byte*, and a byte can be a letter, a number, or a special character such as an exclamation point or a question mark. If your computer has 640,000 bytes of RAM, it can hold a total of 640,000 characters.

All the zeros following RAM measurements can become cumbersome. You often see the shortcut notation *K* (which comes from the metric system's *kilo*, meaning *1000*) in place of the last three zeros. In computer terms, *K* means exactly 1024 bytes; but this number is usually rounded to 1000 to make it easier to remember. Therefore, 640K represents approximately 640,000 bytes of RAM. For more information, see the sidebar titled "The Power of Two."

The limitations of RAM are similar to the limitations of audio cassette tapes. If a cassette is manufactured to hold 60 minutes of

music, it cannot hold 75 minutes of music. Likewise, the total number of characters that compose your program, the C++ data, and your computer's system programs cannot exceed the RAM's limit (unless you save some of the characters to disk).

You want as much RAM as possible to hold C++, data, and the system programs. Generally, 640K is ample room for anything you might want to do in C++. Computer RAM is relatively inexpensive, so if your computer has less than 640K bytes of memory, you should consider purchasing additional memory to increase the total RAM to 640K. You can put more than 640K in most PCs. There are two types of additional RAM: *extended* memory and *expanded* memory (they both offer memory capacity greater than 640K). You can access this extra RAM with some C++ systems, but most beginning C++ programmers have no need to worry about RAM beyond 640K.

The Power of Two

Although K means approximately 1000 bytes of memory, K equates to 1024. Computers function using *on* and *off* states of electricity. These are called *binary* states. At the computer's lowest level, it does nothing more than turn electricity on and off with many millions of switches called *transistors*. Because these switches have two possibilities, the total number of states of these switches—and thus the total number of states of electricity—equals a number that is a power of 2.

The closest power of 2 to 1000 is 1024 (2 to the 10th power). The inventors of computers designed memory so that it is always added in kilobytes, or multiples of 1024 bytes at a time. Therefore, if you add 128K of RAM to a computer, you are actually adding a total of 131,072 bytes of RAM (128 times 1024 equals 131,072).

Because K actually means *more* than 1000, you always have a little more memory than you bargained for! Even though your computer might be rated at 640K, it actually holds more than 640,000 bytes (655,360 to be exact). See Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," for a more detailed discussion of memory.

The computer stores C++ programs to RAM as you write them. If you have used a word processor before, you have used RAM. As you type words in your word-processed documents, your words appear on the video screen and also go to RAM for storage.

Despite its importance, RAM is only one type of memory in your computer. RAM is *volatile*; when you turn the computer off, all RAM is erased. Therefore, you must store the contents of RAM to a nonvolatile, more permanent memory device (such as a disk) before you turn off your computer. Otherwise, you lose your work.

Disk Storage

A *disk* is another type of computer memory, sometimes called *external memory*. Disk storage is nonvolatile. When you turn off your computer, the disk's contents do not go away. This is important. After typing a long C++ program in RAM, you do not want to retype the same program every time you turn your computer back on. Therefore, after creating a C++ program, you save the program to disk, where it remains until you're ready to retrieve it again.

Disk storage differs from RAM in ways other than volatility. Disk storage cannot be processed by the CPU. If you have a program or data on disk that you want to use, you must transfer it from the disk to RAM. This is the only way the CPU can work with the program or data. Luckily, most disks hold many times more data than the RAM's 640K. Therefore, if you fill up RAM, you can store its contents on disk and continue working. As RAM continues to fill up, you or your C++ program can keep storing the contents of RAM to the disk.

This process might sound complicated, but you have only to understand that data must be transferred to RAM before your computer can process it, and saved to disk before you shut your computer off. Most the time, a C++ program runs in RAM and retrieves data from the disk as it needs it. In Chapter 30, "Sequential Files," you learn that working with disk files is not difficult.

There are two types of disks: *hard disks* and *floppy disks*. Hard disks (sometimes called *fixed disks*) hold much more data and are many times faster to work with than floppy disks. Most of your C++ programs and data should be stored on your hard disk. Floppy disks

EXAMPLE

are good for backing up hard disks, and for transferring data and programs from one computer to another. (These removable floppy disks are often called *diskettes*.) Figure 1.2 shows two common sizes, the 5 1/4-inch disk and the 3 1/2-inch disk. These disks can hold from 360K to 1.4 million bytes of data.

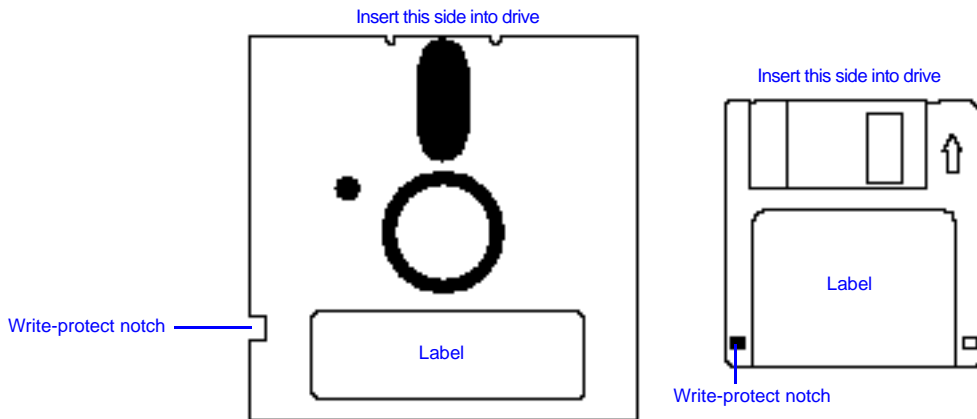


Figure 1.2. 5 1/4-inch disk and 3 1/2-inch disk.

Before using a new box of disks, you have to format them (unless you buy disks that are already formatted). Formatting prepares the disks for use on your computer by writing a pattern of paths, called *tracks*, where your data and programs are stored. Refer to the operating system instruction manual for the correct formatting procedure.

Disk drives house the disks in your computer. Usually, the disk drives are stored in your system unit. The hard disk is sealed inside the hard disk drive, and you never remove it (except for repairs). In general, the floppy disk drives also are contained in the system unit, but you insert and remove these disks manually.

Disk drives have names. The computer's first floppy disk drive is called drive A. The second floppy disk drive, if you have one, is called drive B. The first hard disk (many computers have only one) is called drive C. If you have more than one hard disk, or if your hard disk is logically divided into more than one, the others are named drive D, drive E, and so on.

Disk size is measured in bytes, just as RAM is. Disks can hold many millions of bytes of data. A 60-million-byte hard disk is common. In computer terminology, a million bytes is called a *megabyte*, or *M*. Therefore, if you have a 60-megabyte hard disk, it can hold approximately 60 million characters of data before it runs out of space.

The Monitor

The television-like screen is called the *monitor*. Sometimes the monitor is called the *CRT* (which stands for the primary component of the monitor, the *cathode-ray tube*). The monitor is one place where the output of the computer can be sent. When you want to look at a list of names and addresses, you could write a C++ program to list the information on the monitor.

The advantage of screen output over printing is that screen output is faster and does not waste paper. Screen output, however, is not permanent. When text is *scrolled* off-screen (displaced by additional text coming on-screen), it is gone and you might not always be able to see it again.

All monitors have a *cursor*, which is a character such as a blinking underline or a rectangle. The cursor moves when you type letters on-screen, and always indicates the location of the next character to be typed.

Monitors that can display pictures are called *graphics monitors*. Most PC monitors are capable of displaying graphics and text, but some can display only text. If your monitor cannot display colors, it is called a *monochrome* monitor.

Your monitor plugs into a *display adapter* located in your system unit. The display adapter determines the amount of resolution and number of possible on-screen colors. *Resolution* refers to the number of row and column intersections. The higher the resolution, the more rows and columns are present on your screen and the sharper your text and graphics appear. Some common display adapters are MCGA, CGA, EGA, and VGA.

The Printer

The printer provides a more permanent way of recording your computer's results. It is the "typewriter" of the computer. Your printer can print C++ program output to paper. Generally, you can print anything that appears on your screen. You can use your printer to print checks and envelopes too, because most types of paper work with computer printers.

The two most common PC printers are the *dot-matrix* printer and the *laser* printer. A dot-matrix printer is inexpensive, fast, and uses a series of small dots to represent printed text and graphics. A laser printer is faster than a dot-matrix, and its output is much sharper because a laser beam burns toner ink into the paper. For many people, a dot-matrix printer provides all the speed and quality they need for most applications. C++ can send output to either type of printer.

The Keyboard

Figure 1.3 shows a typical PC keyboard. Most the keys are the same as those on a standard typewriter. The letter and number keys in the center of the keyboard produce their indicated characters on-screen. If you want to type an uppercase letter, be sure to press one of the Shift keys before typing the letter. Pressing the CapsLock key shifts the keyboard to an uppercase mode. If you want to type one of the special characters above a number, however, you must do so with the Shift key. For instance, to type the percent sign (%), you would press Shift-5.

Like the Shift keys, the Alt and Ctrl keys can be used with some other keys. Some C++ programs require that you press Alt or Ctrl before pressing another key. For instance, if your C++ program prompts you to press Alt-F, you should press the Alt key, then press F while still holding down Alt, then release both keys. Do not hold them both down for long, however, or the computer keeps repeating your keystrokes as if you typed them more than once.

The key marked Esc is called the *escape* key. In many C++ programs, you can press Esc to "escape," or exit from, something you started and then wanted to stop. For example, if you prompt your C++ compiler for help and you no longer need the help

message, you can press Esc to remove the help message from the screen.

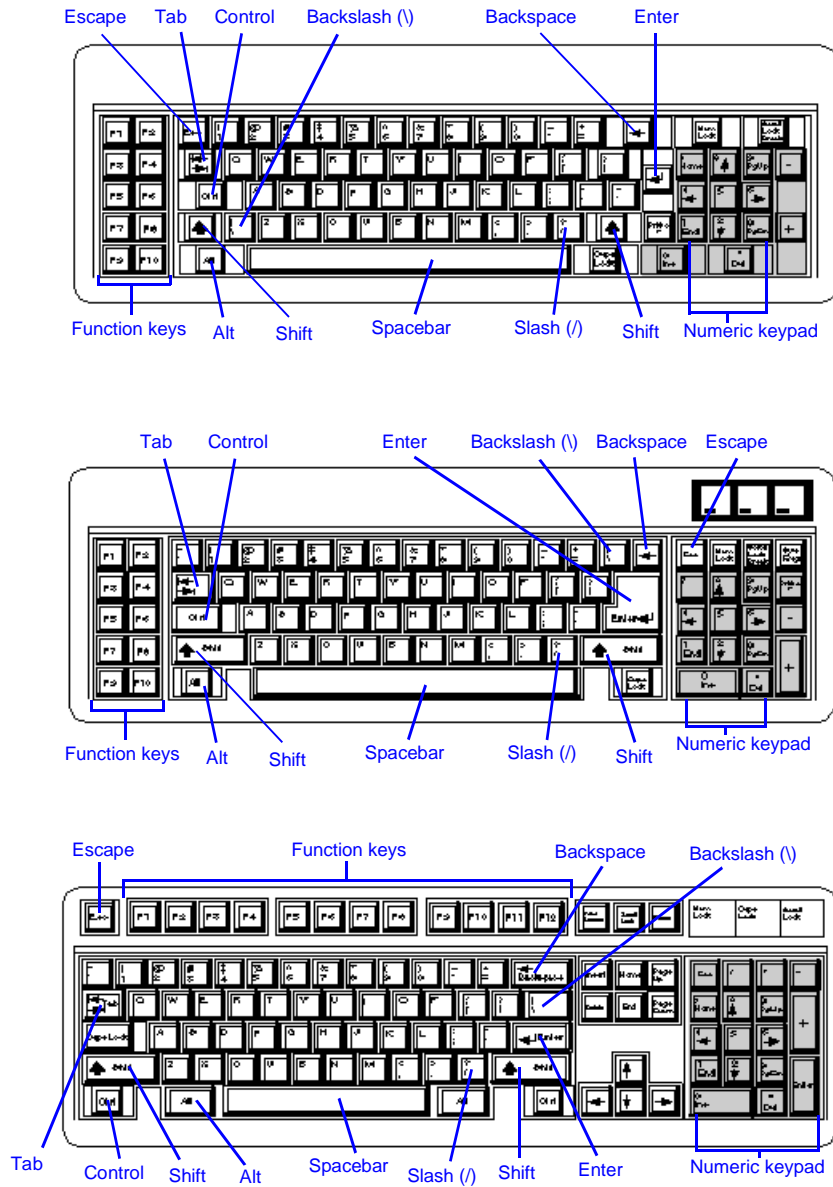


Figure 1.3. The various PC keyboards.

The group of numbers and arrows on the far right of the keyboard is called the *numeric keypad*. People familiar with a 10-key adding machine usually prefer to type numbers from the keypad rather than from the top row of the alphabetic key section. The numbers on the keypad work only when you press the NumLock key. If you press NumLock a second time, you disable these number keys and make the arrow keys work again. To prevent confusion, many keyboards have separate arrow keys and a keypad used solely for numbers.

The arrows help you move the cursor from one area of the screen to another. To move the cursor toward the top of the screen, you have to press the up arrow continuously. To move the cursor to the right, you press the right-arrow, and so on. Do not confuse the Backspace key with the left-arrow. Pressing Backspace moves the cursor backward one character at a time—erasing everything as it moves. The left-arrow simply moves the cursor backward, without erasing.

The keys marked Insert and Delete (Ins and Del on some keyboards) are useful for editing. Your C++ program editor probably takes advantage of these two keys. Insert and Delete work on C++ programs in the same way they work on a word processor's text. If you do not have separate keys labeled Insert and Delete, you probably have to press NumLock and use the keypad key 0 (for Insert) and period (for Delete).

PgUp and PgDn are the keys to press when you want to scroll the screen (that is, move your on-screen text either up or down). Your screen acts like a camera that pans up and down your C++ programs. You can move the screen down your text by pressing PgDn, and up by pressing PgUp. (Like Insert and Delete, you might have to use the keypad for these operations.)

The keys labeled F1 through F12 (some keyboards go only to F10) are called *function keys*. The function keys are located either across the top of the alphabetic section or to the left of it. These keys perform an advanced function, and when you press one of them, you usually want to issue a complex command, such as searching for a specific word in a program. The function keys in your C++ program, however, do not necessarily produce the same results as they might in another program, such as a word processor. In other words, function keys are *application-specific*.



CAUTION: Computer keyboards have a key for number 1, so do not substitute the lowercase *l* to represent the number 1, as you might on a typewriter. To C++, a *l* is different from the letter *l*. You should be careful also to use *0* when you mean zero, and *O* when you want the uppercase letter *O*.

The Mouse

The mouse is a relatively new input device. The mouse moves the cursor to any on-screen location. If you have never used a mouse before, you should take a little time to become skillful in moving the cursor with it. Your C++ editor (described in Chapter 2, “What is a Program?”) might use the mouse for selecting commands from its menus.

Mouse devices have two or three buttons. Most of the time, pressing the third button produces the same results as simultaneously pressing both keys on a two-button mouse.

The Modem

A modem can be used to communicate between two distant computers.

A PC *modem* enables your PC to communicate with other computers over telephone lines. Some modems, called *external modems*, sit in a box outside your computer. *Internal modems* reside inside the system unit. It does not matter which one you have, because they operate identically.

Some people have modems so they can share data between their computer and that of a long-distance friend or off-site coworker. You can write programs in C++ that communicate with your modem.

A Modem by Any Other Name...

The term *digital computer* comes from the fact that your computer operates on binary (on and off) digital impulses of electricity. These digital states of electricity are perfect for your computer's equipment, but they cannot be sent over normal telephone lines. Telephone signals are called *analog* signals, which are much different from the binary digital signals in your PC.

Therefore, before your computer can transmit data over a telephone line, the information must be *modulated* (converted) to analog signals. The receiving computer must have a way to *demodulate* (convert back) those signals to digital.

The modem is the means by which computer signals are modulated and demodulated from digital to analog and vice versa. Thus, the name of the device that *modulates* and *demodulates* these signals is *modem*.

Software

No matter how fast, large, and powerful your computer's hardware is, its software determines what work is done and how the computer does it. Software is to a computer what music is to a stereo system. You store software on the computer's disk and load it in your computer's memory when you are ready to process the software, just as you store music on a tape and play it when you want to hear music.

Programs and Data

No doubt you have heard the phrase, *data processing*. This is what computers actually do: they take data and manipulate it into

meaningful output. The meaningful output is called *information*. Figure 1.4 shows the *input-process-output* model, which is the foundation of everything that happens in your computer.

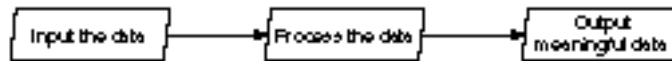


Figure 1.4. Data processing at its most elementary level.

In Chapter 2, “What Is a Program?,” you learn the mechanics of programs. For now, you should know that the programs you write in C++ process the data that you input in the programs. Both data and programs compose the software. The hardware acts as a vehicle to gather the input and produce the output. Without software, computers would be worthless, just as an expensive stereo would be useless without some way of playing music so you can hear it.

The input comes from input devices, such as keyboards, modems, and disk drives. The CPU processes the input and sends the results to the output devices, such as the printer and the monitor. A C++ payroll program might receive its input (the hours worked) from the keyboard. It would instruct the CPU to calculate the payroll amounts for each employee in the disk files. After processing the payroll, the program could print the checks.

MS-DOS

MS-DOS (Microsoft disk operating system) is a system that lets your C++ programs interact with hardware. MS-DOS (commonly called DOS) is always loaded into RAM when you turn on your computer. DOS controls more than just the disks; DOS is there so your programs can communicate with all the computer’s hardware, including the monitor, keyboard, and printer.

Figure 1.5 illustrates the concept of DOS as the “go-between” with your computer’s hardware and software. Because DOS understands how to control every device hooked to your computer, it stays in RAM and waits for a hardware request. For instance, printing the words “C++ is fun!” on your printer takes many computer instructions. However, you do not have to worry about all

those instructions. When your C++ program wants to print something, it tells DOS to print it. DOS always knows how to send information to your printer, so it takes your C++ program requests and does the work of routing that data to the printer.

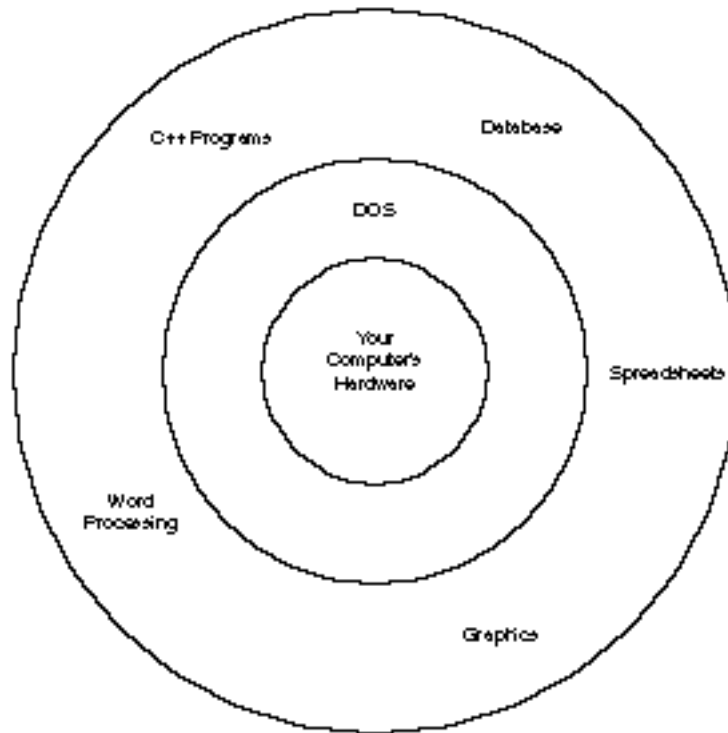


Figure 1.5. DOS interfaces between hardware and software.

Many people program computers for years and never take the time to learn why DOS is there. You do not have to be an expert in DOS, or even know more than a few simple DOS commands, to be proficient with your PC. Nevertheless, DOS does some things that C++ cannot do, such as formatting disks and copying files to your disks. As you learn more about the computer, you might see the need to better understand DOS. For a good introduction to using DOS, refer to the book *MS-DOS 5 QuickStart* (Que).



NOTE: As mentioned, DOS always resides in RAM and is loaded when you start the computer. This is done automatically, so you can use your computer and program in C++ without worrying about how to transfer DOS to RAM. It is important to remember that DOS always uses some of your total RAM.

Figure 1.6 shows you the placement of DOS, C++, and your C++ data area in RAM. This formation is a typical way to represent RAM—several boxes stacked on top of each other. Each memory location (each byte) has a unique *address*, just as everybody’s residence has a unique address. The first address in memory begins at 0, the second RAM address is 1, and so on until the last RAM location, many thousands of bytes later.

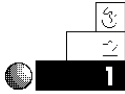


Figure 1.6. After MS-DOS and a C++ program, there is less RAM for data.

Your operating system (whether you use MS-DOS, PC DOS, DR DOS, or UNIX) takes part of the first few thousand bytes of memory. The amount of RAM that DOS takes varies with each computer’s configuration. When working in C++, the C++ system sits on top of DOS, leaving you with the remainder of RAM for your program and data. This explains why you might have a total of 512K of RAM and still not have enough memory to run some programs—DOS is using some of the RAM for itself.

Review Questions

The answers to each chapter's review questions are in Appendix B, aptly named "Answers to Review Questions."



1. What is the name of one of the programming languages from which C was developed?
2. True or false: C++ is known as a "better C."
3. In what decade was C++ developed?
4. True or false: C++ is too large to fit on many micro-computers.
5. Which usually holds more data: RAM or the hard disk?
6. What device is needed for your PC to communicate over telephone lines?



7. Which of the following device types best describes the mouse?
 - a. Storage
 - b. Input
 - c. Output
 - d. Processing
8. What key would you press to turn off the numbers on the numeric keypad?



9. What operating system is written almost entirely in C?
10. Why is RAM considered volatile?
11. True or false: The greater the resolution, the better the appearance of graphics on-screen.
12. How many bytes is 512K?
13. What does *modem* stand for?

Summary

C++ is an efficient, powerful, and popular programming language. Whether you are new to C++ or an experienced programmer, C++ is all you need to program the computer to work the way you want it to.

This chapter presented the background of C++ by walking you through the history of its predecessor, the C programming language. C++ adds to C and offers some of the most advanced programming language commands that exist today.

The rest of this book is devoted to teaching you C++. Chapter 2, “What Is a Program?,” explains program concepts so you can begin to write C++ programs.

What Is a Program?

This chapter introduces you to fundamental programming concepts. The task of programming computers has been described as rewarding, challenging, easy, difficult, fast, and slow. Actually, it is a combination of all these descriptions. Writing complex programs to solve advanced problems can be frustrating and time-consuming, but you can have fun along the way, especially with the rich assortment of features that C++ has to offer.

This chapter also describes the concept of programming, from a program's inception to its execution on your computer. The most difficult part of programming is breaking the problem into logical steps that the computer can execute. Before you finish this chapter, you will type and execute your first C++ program.

This chapter introduces you to

- ◆ The concept of programming
- ◆ The program's output
- ◆ Program design
- ◆ Using an editor
- ◆ Using a compiler

- ♦ Typing and running a C++ program
- ♦ Handling errors

After you complete this chapter, you should be ready to learn the C++ programming language elements in greater detail.

Computer Programs

Before you can make C++ work for you, you must write a C++ program. You have seen the word *program* used several times in this book. The following note defines a program more formally.



NOTE: A *program* is a list of instructions that tells the computer to do things.

Keep in mind that computers are only machines. They're not smart; in fact, they're quite the opposite! They don't do anything until they are given detailed instructions. A word processor, for example, is a program somebody wrote—in a language such as C++—that tells your computer exactly how to behave when you type words into it.

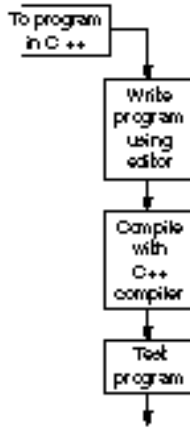
You are familiar with the concept of programming if you have ever followed a recipe, which is a “program,” or a list of instructions, telling you how to prepare a certain dish. A good recipe lists these instructions in their proper order and with enough description so you can carry out the directions successfully, without assuming anything.

If you want your computer to help with your budget, keep track of names and addresses, or compute your gas mileage, it needs a program to tell it how to do those things. You can supply that program in two ways: buy a program somebody else wrote, or write the program yourself.

Writing the program yourself has a big advantage for many applications: The program does exactly what *you* want it to do. If you buy one that is already written, you have to adapt your needs to those of the author of the program. This is where C++ comes into

play. With the C++ programming language (and a little studying), you can make your computer carry out your own tasks precisely.

To give C++ programming instructions to your computer, you need an *editor* and a *C++ compiler*. An editor is similar to a word processor; it is a program that enables you to type a C++ program into memory, make changes (such as moving, copying, inserting, and deleting text), and save the program more permanently in a disk file. After you use the editor to type the program, you must compile it before you can run it.



The C++ programming language is called a *compiled* language. You cannot write a C++ program and run it on your computer unless you have a C++ compiler. This compiler takes your C++ language instructions and translates them into a form that your computer can read. A C++ compiler is the tool your computer uses to understand the C++ language instructions in your programs. Many compilers come with their own built-in editor. If yours does, you probably feel that your C++ programming is more integrated.

To some beginning programmers, the process of compiling a program before running it might seem like an added and meaningless step. If you know the BASIC programming language, you might not have heard of a compiler or understand the need for one. That's because BASIC (also APL and some versions of other computer languages) is not a compiled language, but an *interpreted* language. Instead of translating the entire program into machine-readable form (as a compiler does in one step), an interpreter translates each program instruction—then executes it—before translating the next one. The difference between the two is subtle, but the bottom line is not: Compilers produce *much* more efficient and faster-running programs than interpreters do. This seemingly extra step of compiling is worth the effort (and with today's compilers, there is not much extra effort needed).

Because computers are machines that do not think, the instructions you write in C++ must be detailed. You cannot assume your computer understands what to do if some instruction is not in your program, or if you write an instruction that does not conform to C++ language requirements.

After you write and compile a C++ program, you have to *run*, or *execute*, it. Otherwise, your computer would not know that you

want it to follow the instructions in the program. Just as a cook must follow a recipe's instructions before making the dish, so too your computer must execute a program's instructions before it can accomplish what you want it to do. When you run a program, you are telling the computer to carry out your instructions.

The Program and Its Output

While you are programming, remember the difference between a program and its output. Your program contains only the C++ instructions that you write, but the computer follows your instructions only *after* you run the program.

Throughout this book, you often see a *program listing* (that is, the C++ instructions in the program) followed by the results that occur when you run the program. The results are the output of the program, and they go to an output device such as the screen, the printer, or a disk file.

Program Design

Design your programs before you type them.

You must plan your programs before typing them into your C++ editor. When builders construct houses, for example, they don't immediately grab their lumber and tools and start building! They first find out what the owner of the house wants, then they draw up the plans, order the materials, gather the workers, and finally start building the house.

The hardest part of writing a program is breaking it into logical steps that the computer can follow. Learning the C++ language is a requirement, but it is not the only thing to consider. There is a method of writing programs, a formal procedure you should learn, that makes your programming job easier. To write a program you should:

1. Define the problem to be solved with the computer.
2. Design the program's output (what the user should see).

3. Break the problem into logical steps to achieve this output.
4. Write the program (using the editor).
5. Compile the program.
6. Test the program to assure it performs as you expect.

As you can see from this procedure, the typing of your program occurs toward the end of your programming. This is important, because you first have to plan *how* to tell the computer how to perform each task.

Your computer can perform instructions only step-by-step. You must assume that your computer has no previous knowledge of the problem, so it is up to you to provide that knowledge, which, after all, is what a good recipe does. It would be a useless recipe for a cake if all it said was: "Bake the cake." Why? Because this *assumes* too much on the part of the baker. Even if you write the recipe in step-by-step fashion, proper care must be taken (through planning) to be sure the steps are in sequence. Wouldn't it be foolish also to instruct a baker to put the ingredients into the oven before stirring them?

This book adheres to the preceding programming procedure throughout the book, as each program appears. Before you see the actual program, the thought process required to write the program appears. The goals of the program are presented first, then these goals are broken into logical steps, and finally the program is written.

Designing the program in advance guarantees that the entire program structure is more accurate and keeps you from having to make changes later. A builder, for example, knows that a room is much harder to add after the house is built. If you do not properly plan every step, it is going to take you longer to create the final, working program. It is always more difficult to make major changes after you write your program.

Planning and developing according to these six steps becomes much more important as you write longer and more complicated programs. Throughout this book, you learn helpful tips for program design. Now it's time to launch into C++, so you can experience the satisfaction of typing your own program and seeing it run.

Using a Program Editor

The instructions in your C++ program are called the *source code*. You type source code into your computer's memory by using your program editor. After you type your C++ source code (your program), you should save it to a disk file before compiling and running the program. Most C++ compilers expect C++ source programs to be stored in files with names ending in .CPP. For example, the following are valid filenames for most C++ compilers:

MYPROG.CPP

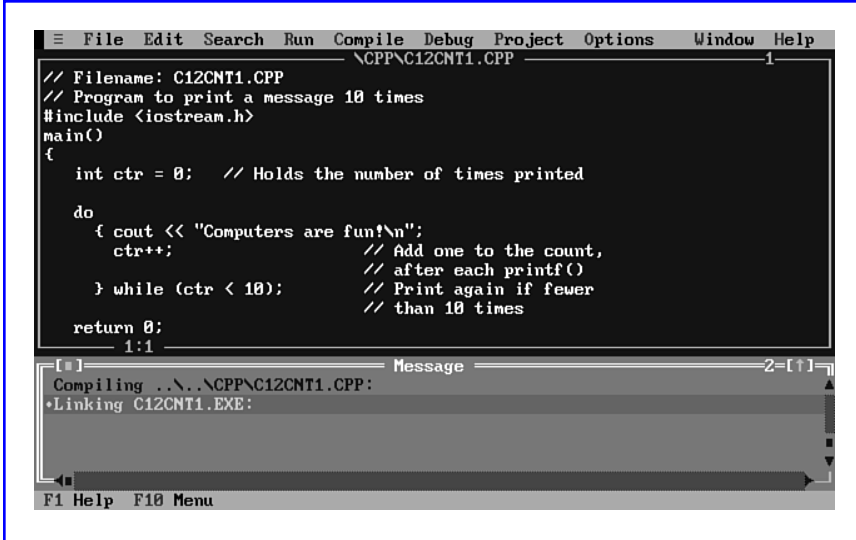
SALESACT.CPP

EMPLYEE.CPP

ACCREC.CPP

Many C++ compilers include a built-in editor. Two of the most popular C++ compilers (both conform to the AT&T C++ 2.1 standard and include their own extended language elements) are Borland's C++ and Microsoft's C/C++ 7.0 compilers. These two programs run in fully integrated environments that relieve the programmer from having to worry about finding a separate program editor or learning many compiler-specific commands.

Figure 2.1 shows a Borland C++ screen. Across the top of the screen (as with Microsoft C/C++ 7.0) is a menu that offers pull-down editing, compiling, and running options. The middle of the screen contains the body of the program editor, and this is the area where the program goes. From this screen, you type, edit, compile, and run your C++ source programs. Without an *integrated environment*, you would have to start an editor, type your program, save the program to disk, exit the editor, run the compiler, and only *then* run the compiled program from the operating system. With Borland's C++ and Microsoft C/C++ 7.0, you simply type the program into the editor, then—in one step—you select the proper menu option that compiles and runs the program.



```
File Edit Search Run Compile Debug Project Options Window Help
\CPP\C12CNT1.CPP
// Filename: C12CNT1.CPP
// Program to print a message 10 times
#include <iostream.h>
main()
{
  int ctr = 0; // Holds the number of times printed

  do
  { cout << "Computers are fun!\n";
    ctr++; // Add one to the count,
           // after each printf()
  } while (ctr < 10); // Print again if fewer
                    // than 10 times

  return 0;
}
1:1

Message
Compiling ..\..\CPP\C12CNT1.CPP:
Linking C12CNT1.EXE:

F1 Help F10 Menu
```

Figure 2.1. Borland Turbo C++'s integrated environment.

If you do not own an integrated environment such as Borland C++ or Microsoft C/C++, you have to find a program editor. Word processors can act as editors, but you have to learn how to save and load files in a true ASCII text format. It is often easier to use an editor than it is to make a word processor work like one.

On PCs, DOS Version 5 comes with a nice, full-screen editor called EDIT. It offers menu-driven commands and full cursor-control capabilities. EDIT is a simple program to use, and is a good beginner's program editor. Refer to your DOS manual or a good book on DOS, such as *MS-DOS 5 QuickStart* (Que), for more information on this program editor.

Another editor, called EDLIN, is available for earlier versions of DOS. EDLIN is a line editor that does not allow full-screen cursor control, and it requires you to learn some cryptic commands. The advantage to learning EDLIN is that it is always included with all PCs that use a release of DOS prior to Version 5.

If you use a computer other than a PC, such as a UNIX-based minicomputer or a mainframe, you have to determine which editors are available. Most UNIX systems include the `vi` editor. If you program on a UNIX operating system, it would be worth your time to learn `vi`. It is to UNIX what EDLIN is to PC operating systems, and is available on almost every UNIX computer in the world.

Mainframe users have other editors available, such as the ISPF editor. You might have to check with your systems department to find an editor accessible from your account.



NOTE: Because this book teaches the generic AT&T C++ standard programming language, no attempt is made to tie in editor or compiler commands—there are too many on the market to cover them all in one book. As long as you write programs specific to the AT&T C++, the tools you use to edit, compile, and run those programs are secondary; your goal of good programming is the result of whatever applications you produce.

Using a C++ Compiler

After you type and edit your C++ program's source code, you have to compile the program. The process you use to compile your program depends on the version of C++ and the computer you are using. Borland C++ and Microsoft C/C++ users need only press Alt-R to compile and run their programs. When you compile programs on most PCs, your compiler eventually produces an *executable* file with a name beginning with the same name as the source code, but ends with an .EXE file extension. For example, if your source program is named GRADEAVG.CPP, the PC would produce a compiled file called GRADEAVG.EXE, which you could execute at the DOS prompt by typing the name `gradeavg`.



NOTE: Each program in this book contains a comment that specifies a recommended filename for the source program. You do not have to follow the file-naming conventions used in this book; the filenames are only suggestions. If you use a mainframe, you have to follow the dataset-naming conventions set up by your system administrator. Each program name in the sample disk (see the order form at the back of the book) matches the filenames of the program listings.

UNIX users might have to use the `cfront` compiler. Most `cfront` compilers actually convert C++ code into regular C code. The C code is then compiled by the system's C compiler. This produces an executable file whose name (by default) is `A.OUT`. You can then run the `A.OUT` file from the UNIX prompt. Mainframe users generally have company-standard procedures for compiling C++ source programs and storing their results in a test account.

Unlike many other programming languages, your C++ program must be routed through a *preprocessor* before it is compiled. The preprocessor reads preprocessor directives that you enter in the program to control the program's compilation. Your C++ compiler automatically performs the preprocessor step, so it requires no additional effort or commands to learn on your part.

You might have to refer to your compiler's reference manuals or to your company's system personnel to learn how to compile programs for your programming environment. Again, learning the programming environment is not as critical as learning the C++ language. The compiler is just a way to transform your program from a source code file to an executable file.

Your program must go through one additional stage after compiling and before running. It is called the *linking*, or the *link editing* stage. When your program is linked, a program called the linker supplies needed runtime information to the compiled program. You can also combine several compiled programs into one executable program by linking them. Most of the time, however,

your compiler initiates the link editing stage (this is especially true with integrated compilers such as Borland C++ and Microsoft C/C++) and you do not have to worry about the process.

Figure 2.2 shows the steps that your C++ compiler and link editor perform to produce an executable program.

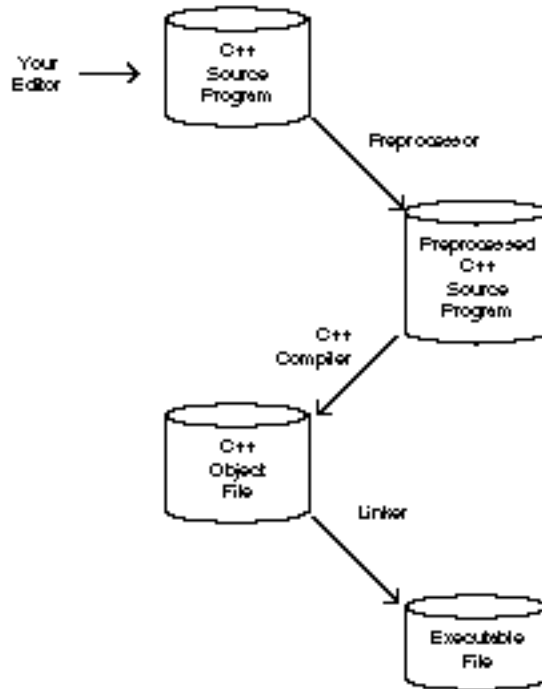


Figure 2.2. Compiling C++ source code into an executable program.

Running a Sample Program

Before delving into the specifics of the C++ language, you should take a few moments to become familiar with your editor and C++ compiler. Starting with the next chapter, “Your First C++ Program,” you should put all your concentration into the C++ programming language and not worry about using a specific editor or compiling environment.

Therefore, start your editor of choice and type Listing 2.1, which follows, into your computer. Be as accurate as possible—a single typing mistake could cause the C++ compiler to generate a series of errors. You do not have to understand the program’s content at this point; the goal is to give you practice in using your editor and compiler.

Listing 2.1. Practicing with the editor.



Comment the program with the program name.

Include the header file `iostream.h` so the output properly works.

Start of the `main()` function.

Define the `BELL` constant, which is the computer’s beep.

Initialize the integer variable `ctr` to 0.

Define the character array `fname` to hold 20 elements.

Print to the screen What is your first name?.

Accept a string from the keyboard.

Process a loop while the variable `ctr` is less than five.

Print the string accepted from the keyboard.

Increment the variable `ctr` by 1.

Print to the screen the character code that sounds the beep.

Return to the operating system.

```
// Filename: C2FIRST.CPP
// Requests a name, prints the name five times, and rings a bell.

#include <iostream.h>

main()
{
    const char BELL='\a';           // Constant that rings the bell
    int ctr=0;                       // Integer variable to count through loop
    char fname[20];                 // Define character array to hold name

    cout << "What is your first name? "; // Prompt the user
    cin >> fname;                     // Get the name from the keyboard
    while (ctr < 5)                  // Loop to print the name
```

```

{
    cout << fname << "\n";
    ctr++;
}
cout << BELL;           // Ring the terminal's bell
return 0;
}

```

Be as accurate as possible. In most programming languages—and especially in C++—the characters you type into a program must be very accurate. In this sample C++ program, for instance, you see parentheses, `()`, brackets, `[]`, and braces, `{}`, but you cannot use them interchangeably.

The *comments* (words following the two slashes, `//`) to the right of some lines do not have to end in the same place that you see in the listing. They can be as long or short as you need them to be. However, you should familiarize yourself with your editor and learn to space characters accurately so you can type this program exactly as shown.

Compile the program and execute it. Granted, the first time you do this you might have to check your reference manuals or contact someone who already knows your C++ compiler. Do not worry about damaging your computer: Nothing you do from the keyboard can harm the physical computer. The worst thing you can do at this point is erase portions of your compiler software or change the compiler's options—all of which can be easily corrected by reloading the compiler from its original source. (It is only remotely likely that you would do anything like this, even if you are a beginner.)

Handling Errors

Because you are typing instructions for a machine, you must be very accurate. If you misspell a word, leave out a quotation mark, or make another mistake, your C++ compiler informs you with an error message. In Borland C++ and Microsoft C/C++, the error probably appears in a separate window, as shown in Figure 2.3. The most common error is a *syntax error*, and this usually implies a misspelled word.

```
File Edit Search Run Compile Debug Project Options Window Help
\\CPP\\C12CNT1.CPP
// Filename: C12CNT1.CPP
// Program to print a message 10 times
#include <iostream.h>
main()
{
  int ctr = 0; // Holds the number of times printed
  do
  { cout << "Computers are fun!\\n";
    ctr++; // Add one to the count,
           // after each printf()
  } while (ctr < 10); // Print again if fewer
                   // than 10 times
  return 0;
}
3:1
Message
Compiling ..\\.\\CPP\\C12CNT1.CPP:
Error ..\\.\\CPP\\C12CNT1.CPP 3: Unable to open include file 'ISTREAM.H'
Error ..\\.\\CPP\\C12CNT1.CPP 9: Undefined symbol 'cout'
F1 Help Space View source Edit source F10 Menu
```

Figure 2.3. The compiler reporting a program error.

When you get an error message (or more than one), you must return to the program editor and correct the error. If you don't understand the error, you might have to check your reference manual or scour your program's source code until you find the offending code line.

Getting the Bugs Out

One of the first computers, owned by the military, refused to print some important data one day. After its programmers tried for many hours to find the problem in the program, a programmer by the name of Grace Hopper decided to check the printer.

She found a small moth lodged between two important wires. When she removed the moth, the printer started working perfectly (although the moth did not have the same luck).

Grace Hopper was an admiral from the Navy and, although she was responsible for developing many important computer concepts (she was the author of the original COBOL language), she might be best known for discovering the first computer bug.

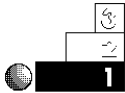
Ever since Admiral Hopper discovered that moth, errors in computer programs have been known as *computer bugs*. When you test your programs, you might have to *debug* them—get the bugs (errors) out by correcting your typing errors or changing the logic so your program does exactly what you want it to do.

After you have typed your program correctly using the editor (and you get no compile errors), the program should run properly by asking for your first name, then printing it on-screen five times. After it prints your name for the fifth time, you hear the computer's bell ring.

This example helps to illustrate the difference between a program and its output. You must type the program (or load one from disk), then run the program to see its output.

Review Questions

The answers to the review questions are in Appendix B, “Answers to Review Questions.”



1. What is a program?
2. What are the two ways to obtain a program that does what you want?
3. True or false: Computers can think.
4. What is the difference between a program and its output?
5. What do you use for typing C++ programs into the computer?



6. What filename extension do all C++ programs have?
7. Why is typing the program one of the *last* steps in the programming process?



8. What does the term *debug* mean?
9. Why is it important to write programs that are compatible with the AT&T C++?
10. True or false: You must link a program before compiling it.

Summary

After reading this chapter, you should understand the steps necessary to write a C++ program. You know that planning makes writing the program much easier, and that your program's instructions produce the output only after you run the program.

You also learned how to use your program editor and compiler. Some program editors are as powerful as word processors. Now that you know how to run C++ programs, it is time to start learning the C++ programming language.

Your First C++ Program

This chapter introduces you to some important C++ language commands and other elements. Before looking at the language more specifically, many people like to “walk through” a few simple programs to get an overall feel for what a C++ program involves. This is done here. The rest of the book covers these commands and elements more formally.

This chapter introduces the following topics:

- ◆ An overview of C++ programs and their structure
- ◆ Variables and literals
- ◆ Simple math operators
- ◆ Screen output format

This chapter introduces a few general tools you need to become familiar with the C++ programming language. The rest of the book concentrates on more specific areas of the actual language.

Looking at a C++ Program

Figure 3.1 shows the outline of a typical small C++ program. No C++ commands are shown in the figure. Although there is much more to a program than this outline implies, this is the general format of the beginning examples in this book.

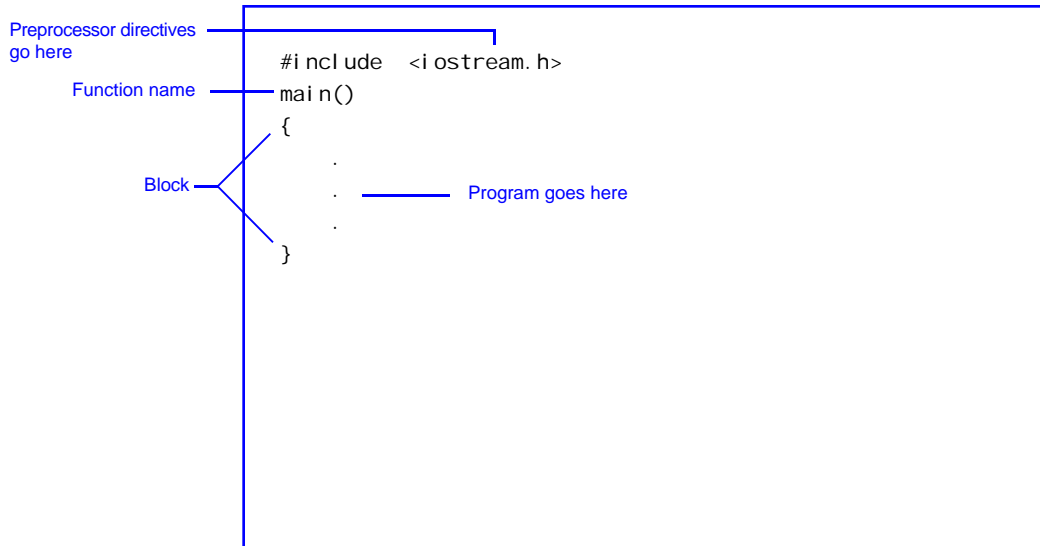


Figure 3.1. A skeleton outline of a simple C++ program.

To acquaint yourself with C++ programs as fast as possible, you should begin to look at a program in its entirety. The following is a listing of a simple example C++ program. It doesn't do much, but it enables you to see the general format of C++ programming. The next few sections cover elements from this and other programs. You might not understand everything in this program, even after finishing the chapter, but it is a good place to start.

```

// Filename: C3FIRST.CPP
// Initial C++ program that demonstrates the C++ comments
// and shows a few variables and their declarations.
  
```

```
#include <iostream.h>

main()
{
    int i, j;    // These three lines declare four variables.
    char c;
    float x;

    i = 4;      // i and j are both assigned integer literals.
    j = i + 7;
    c = 'A';    // All character literals are
                // enclosed in single quotations.
    x = 9.087;  // x requires a floating-point value because it
                // was declared as a floating-point variable.
    x = x * 4.5; // Change what was in x with a formula.

    // Sends the values of the variables to the screen.
    cout << i << ", " << j << ", " << c << ", " << x << "\n";

    return 0;   // ALWAYS end programs and functions with return.
                // The 0 returns to the operating system and
                // usually indicates no errors occurred.
}
```

For now, familiarize yourself with this overall program. See if you can understand any part or all of it. If you are new to programming, you should know that the computer reads each line of the program, starting with the first line and working its way down, until it has completed all the instructions in the program. (Of course, you first have to compile and link the program, as described in Chapter 2, “What Is a Program?”.)

The output of this program is minimal: It simply displays four values on-screen after performing some assignments and calculations of arbitrary values. Just concentrate on the general format at this point.

The Format of a C++ Program

C++ is a free-form language.

Unlike some other programming languages, such as COBOL, C++ is a *free-form* language, meaning that programming statements

can start in any column of any line. You can insert blank lines in a program if you want. This sample program is called C3FIRST.CPP (you can find the name of each program in this book in the first line of each program listing). It contains several blank lines to help separate parts of the program. In a simple program such as this, the separation is not as critical as it might be in a longer, more complex program.

Generally, spaces in C++ programs are free-form as well. Your goal should not be to make your programs as compact as possible. Your goal should be to make your programs as readable as possible. For example, the C3FIRST.CPP program shown in the previous section could be rewritten as follows:

```
// Filename: C3FIRST.CPP Initial C++ program that demonstrates
// the C++ comments and shows a few variables and their
// declarations.
#include <iostream.h>
main(){int i,j; // These three lines declare four variables.
char c; float x; i=4; // i and j are both assigned integer literals.
j=i+7; c='A'; // All character literals are enclosed in
//single quotations.
x=9.087; //x requires a floating-point value because it was
//declared as a floating-point variable.
x=x*4.5; //Change what was in x with a formula.
//Sends the values of the variables to the screen.
cout<<i<<"", "<<j<<"", "<<c<<"", "<<x<<"\n"; return 0; // ALWAYS
//end programs and functions with return. The 0 returns to
//the operating system and usually indicates no errors occurred.
}
```

To your C++ compiler, the two programs are exactly the same, and they produce exactly the same result. However, to people who have to read the program, the first style is much more readable.

Readability Is the Key

As long as programs do their job and produce correct output, who cares how well they are written? Even in today's world of fast computers and abundant memory and disk space, you should still

care. Even if nobody else ever looks at your C++ program, you might have to change it at a later date. The more readable you make your program, the faster you can find what needs changing, and change it accordingly.

If you work as a programmer for a corporation, you can almost certainly expect to modify someone else's source code, and others will probably modify yours. In programming departments, it is said that long-term employees write readable programs. Given this new global economy and all the changes that face business in the years ahead, companies are seeking programmers who write for the future. Programs that are straightforward, readable, abundant with *white space* (separating lines and spaces), and devoid of hard-to-read "tricks" that create messy programs are the most desirable.

Use ample white space so you can have separate lines and spaces throughout your programs. Notice the first few lines of C3FIRST.CPP start in the first column, but the body of the program is indented a few spaces. This helps programmers "zero in" on the important code. When you write programs that contain several sections (called *blocks*), your use of white space helps the reader's eye follow and recognize the next indented block.

Uppercase Versus Lowercase

Use lowercase abundantly in C++!

Your uppercase and lowercase letters are much more significant in C++ than in most other programming languages. You can see that most of C3FIRST.CPP is in lowercase. The entire C++ language is in lowercase. For example, you must type the keywords `int`, `char`, and `return` in programs using lowercase characters. If you use uppercase letters, your C++ compiler would produce many errors and refuse to compile the program until you correct the errors. Appendix E, "Keyword and Function Reference," shows a list of every command in the C++ programming language. You can see that none of the commands have uppercase letters.

Many C++ programmers reserve uppercase characters for some words and messages sent to the screen, printer, or disk file; they use lowercase letters for almost everything else. There is, however, one exception to this rule in Chapter 4, "Variables and Literals," dealing with the `const` keyword.

Braces and `main()`

All C++ programs require the following lines:

```
main()
{
```

The statements that follow `main()` are executed first. The section of a C++ program that begins with `main()`, followed by an opening brace, `{`, is called the *main function*. A C++ program is actually a collection of functions (small sections of code). The function called `main()` is always required and always the first *function* executed.

In the sample program shown here, almost the entire program is `main()` because the matching closing brace that follows `main()`'s opening brace is at the end of the program. Everything between two matching braces is called a *block*. You read more about blocks in Chapter 16, "Writing C++ Functions." For now, you only have to realize that this sample program contains just one function, `main()`, and the entire function is a single block because there is only one pair of braces.

All *executable* C++ statements must have a semicolon (`;`) after them so C++ is aware that the statement is ending. Because the computer ignores all comments, do *not* put semicolons after your comments. Notice that the lines containing `main()` and braces do not end with semicolons either, because these lines simply define the beginning and ending of the function and are not executed.

As you become better acquainted with C++, you learn when to include the semicolon and when to leave it off. Many beginning C++ programmers learn quickly when semicolons are required; your compiler certainly lets you know if you forget to include a semicolon where one is needed.

Figure 3.2 repeats the sample program shown in Figure 3.1. It contains additional markings to help acquaint you with these new terms as well as other items described in the remainder of this chapter.

A C++ block is enclosed in two braces.

All executable C++ statements must end with a semicolon (`;`).

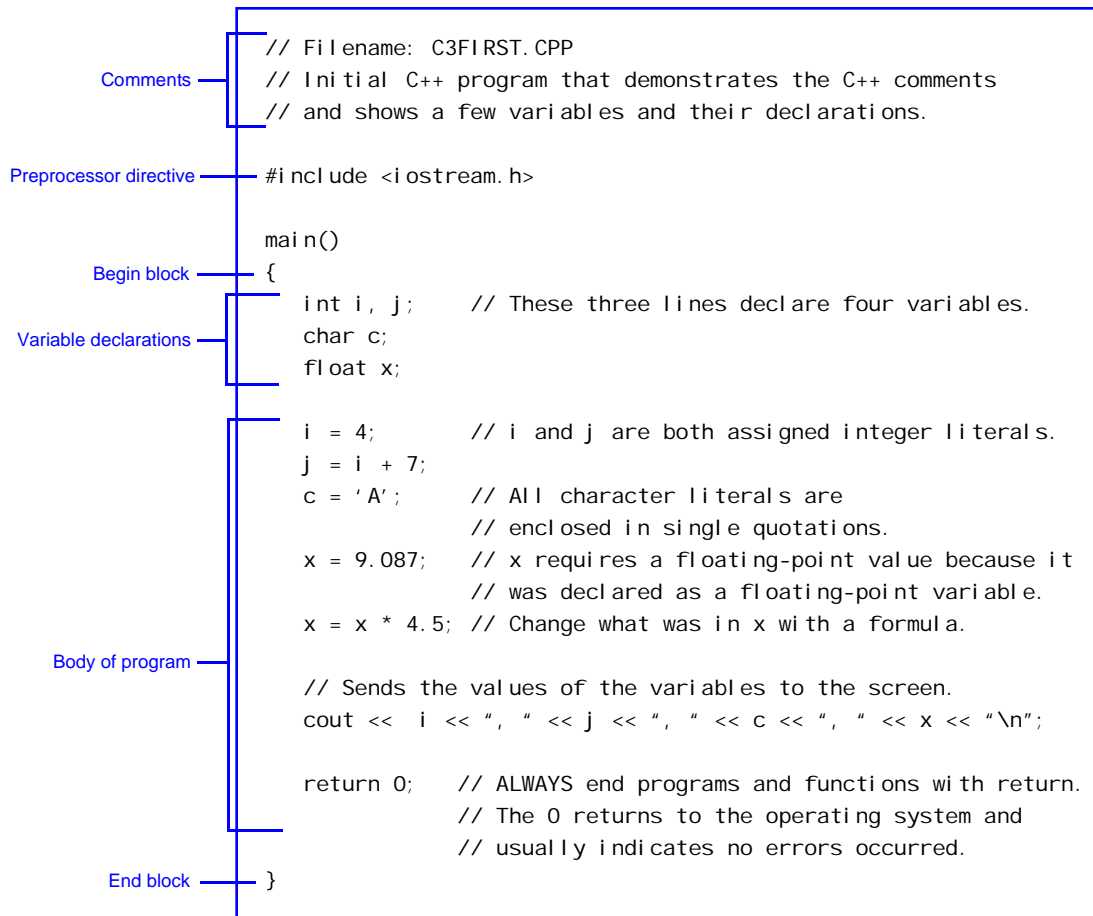


Figure 3.2. The parts of the sample program.

Comments in C++

In Chapter 2, “What Is a Program?,” you learned the difference between a program and its output. Most users of a program do not see the actual program; they see the output from the execution of the program’s instructions. Programmers, on the other hand, look at the program listings, add new routines, change old ones, and update for advancements in computer equipment.



Comments tell people what the program is doing.



As explained earlier, the readability of a program is important so you and other programmers can look through it easily. Nevertheless, no matter how clearly you write C++ programs, you can always enhance their readability by adding comments throughout.

Comments are messages that you insert in your C++ programs, explaining what is going on at that point in the program. For example, if you write a payroll program, you might put a comment before the check-printing routine that describes what is about to happen. You never put C++ language statements inside a comment, because a comment is a message for people—not computers. Your C++ compiler ignores all comments in every program.

NOTE: C++ comments always begin with a // symbol and end at the end of the line.

Some programmers choose to comment several lines. Notice in the sample program, C3FIRST.CPP, that the first three lines are comment lines. The comments explain the filename and a little about the program.

Comments also can share lines with other C++ commands. You can see several comments sharing lines with commands in the C3FIRST.CPP program. They explain what the individual lines do. Use abundant comments, but remember who they're for: people, not computers. Use comments to help explain your code, but do not *overcomment*. For example, even though you might not be familiar with C++, the following statement is easy: It prints “C++ By Example” on-screen.



```
cout << "C++ By Example"; // Print C++ By Example on-screen.
```

This comment is redundant and adds nothing to your understanding of the line of code. It would be much better, in this case, to leave out the comment. If you find yourself almost repeating the C++ code, leave out that particular comment. Not every line of a C++ program should be commented. Comment only when code lines need explaining—in English—to the people looking at your program.

It does not matter if you use uppercase, lowercase, or a mixture of both in your comments because C++ ignores them. Most C++

programmers capitalize the first letter of sentences in comments, just as you would in everyday writing. Use whatever case seems appropriate for the letters in your message.

C++ can also use C-style comments. These are comments that begin with `/*` and end with `*/`. For instance, this line contains a comment in the C *and* C++ style:

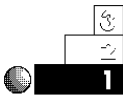
```
netpay = grosspay - taxes; /* Compute take-home pay. */
```

Comment As You Go

Insert your comments as you write your programs. You are most familiar with your program logic at the time you are typing the program in the editor. Some people put off adding comments until after the program is written. More often than not, however, those comments are never added, or else they are written halfheartedly.

If you comment as you write your code, you can glance back at your comments while working on later sections of the program—instead of having to decipher the previous code. This helps you whenever you want to search for something earlier in the program.

Examples



1. Suppose you want to write a C++ program that produces a fancy boxed title containing your name with flashing dots around it (like a marquee). The C++ code to do this might be difficult to understand. Before such code, you might want to insert the following comment so others can understand the code later:

```
// The following few lines draw a fancy box around
// a name, then display flashing dots around the
// name like a Hollywood movie marquee.
```

This would not tell C++ to do anything because a comment is not a command, but it would make the next few lines of code more understandable to you and others. The comment explains in English, for people reading the program, exactly what the program is getting ready to do.



2. You should also put the disk filename of the program in one of the first comments. For example, in the C3FIRST.CPP program shown earlier, the first line is the beginning of a comment:

```
// Filename: C3FIRST.CPP
```

The comment is the first of three lines, but this line tells you in which disk file the program is stored. Throughout this book, programs have comments that include a possible filename under which the program can be stored. They begin with Cx, where x is the chapter number in which they appear (for example, C6VARPR.CPP and C10LNIN.CPP). This method helps you find these programs when they are discussed in another section of the book.



TIP: It might be a good idea to put your name at the top of a program in a comment. If people have to modify your program at a later date, they first might want to consult with you, as the original programmer, before they change it.

Explaining the Sample Program

Now that you have an overview of a C++ program, its structure, and its comments, the rest of this chapter walks you through the entire sample program. Do not expect to become a C++ expert just by completing this section—that is what the rest of the book is for! For now, just sit back and follow this step-by-step description of the program code.

As described earlier, this sample program contains several comments. The first three lines of the program are comments:

```
// Filename: C3FIRST.CPP
// Initial C++ program that demonstrates the C++ comments
// and shows a few variables and their declarations.
```

This comment lists the filename and explains the purpose of the program. This is not the only comment in the program; others appear throughout the code.

The next line beginning with `#include` is called a preprocessor directive and is shown here:

```
#include <iostream.h>
```

This strange looking statement is not actually a C++ command, but is a directive that instructs the C++ compiler to load a file from disk into the middle of the current program. The only purpose for this discussion is to ensure that the output generated with `cout` works properly. Chapter 6, “Preprocessor Directives,” more fully explains this directive.

The next two lines (following the blank separating line) are shown here:

```
main()
{
```

This begins the `main()` function. Basically, the `main()` function’s opening and closing braces enclose the body of this program and `main()`’s instructions that execute. C++ programs often contain more than one function, but they *always* contain a function called `main()`. The `main()` function does not have to be the first one, but it usually is. The opening brace begins the first and only block of this program.

When a programmer compiles and runs this program, the computer looks for `main()` and starts executing whatever instruction follows `main()`’s opening brace. Here are the three lines that follow:

```
int i, j; // These three lines declare four variables.
char c;
float x;
```

These three lines declare variables. A *variable declaration* describes variables used in a block of code. Variable declarations describe the program's data storage.

A C++ program processes data into meaningful results. All C++ programs include the following:

- ♦ Commands
- ♦ Data

Data comprises *variables* and *literals* (sometimes called constants). As the name implies, a *variable* is data that can change (become variable) as the program runs. A literal remains the same. In life, a variable might be your salary. It increases over time (if you are lucky). A literal would be your first name or social security number, because each remains with you throughout life and does not (naturally) change.

Chapter 4, "Variables and Literals," fully explains these concepts. However, to give you an overview of the sample program's elements, the following discussion explains variables and literals in this program.

C++ enables you to use several kinds of literals. For now, you simply have to understand that a C++ literal is any number, character, word, or phrase. The following are all valid C++ literals:

5.6
-45
'Q'
"Mary"
18.67643
0.0

As you can see, some literals are numeric and some are character-based. The single and double quotation marks around two of the literals, however, are not part of the actual literals. A single-character literal requires single quotation marks around it; a string of characters, such as "Mary", requires double quotation marks.

EXAMPLE

Look for the literals in the sample program. You find these:

```
4
7
'A'
9.087
4.5
```

A variable is like a box inside your computer that holds something. That “something” might be a number or a character. You can have as many variables as needed to hold changing data. After you define a variable, it keeps its value until you change it or define it again.

Variables have names so you can tell them apart. You use the assignment operator, the equal sign (=), to assign values to variables. The following statement,



```
sal es=25000;
```

puts the literal value 25000 into the variable named `sal es`. In the sample program, you find the following variables:

```
i
j
c
x
```

The three lines of code that follow the opening brace of the sample program declare these variables. This variable declaration informs the rest of the program that two integer variables named `i` and `j` as well as a character variable called `c` and a floating-point variable called `x` appear throughout the program. The terms *integer* and *floating-point* basically refer to two different types of numbers: Integers are whole numbers, and floating-point numbers contain decimal points.

The next few statements of the sample program assign values to these variables.

```

i = 4;           // i and j are both assigned integer literals.
j = i + 7;
c = 'A';        // All character literals are
                // enclosed in single quotations.
x = 9.087;      // x requires a floating-point value because it
                // was declared as a floating-point variable.
x = x * 4.5;    // Change what was in x with a formula.

```

The first line puts 4 in the integer variable, *i*. The second line adds 7 to the variable *i*'s value to get 11, which then is assigned to (or put into) the variable called *j*. The plus sign (+) in C++ works just like it does in mathematics. The other primary math operators are shown in Table 3.1.

Table 3.1. The primary math operators.

<i>Operator</i>	<i>Meaning</i>	<i>Example</i>
+	Addition	4 + 5
-	Subtraction	7 - 2
*	Multiplication	12 * 6
/	Division	48 / 12

The character literal *A* is assigned to the *c* variable. The number 9.087 is assigned to the variable called *x*, then *x* is immediately overwritten with a new value: itself (9.087) multiplied by 4.5. This helps illustrate why computer designers use an asterisk (*) for multiplication and not a lowercase *x* as people generally do to show multiplication; the computer would confuse the variable *x* with the multiplication symbol, *x*, if both were allowed.



TIP: If mathematical operators are on the right side of the equal sign, the program completes the math before assigning the result to a variable.

The next line (after the comment) includes the following special—and, at first, confusing—statement:

```
cout << i << ", " << j << ", " << c << ", " << x << "\n";
```

When the program reaches this line, it prints the contents of the four variables on-screen. The important part of this line is that the four values for `i`, `j`, `c`, and `x` print on-screen.

The output from this line is

```
4, 11, A, 40.891499
```

Because this is the only `cout` in the program, this is the only output the sample program produces. You might think the program is rather long for such a small output. After you learn more about C++, you should be able to write more useful programs.

The `cout` is not a C++ command. You might recall from Chapter 2, “What Is a Program?,” that C++ has no built-in input/output commands. The `cout` is an operator, described to the compiler in the `#include` file called `iostream.h`, and it sends output to the screen.

C++ also supports the `printf()` function for formatted output. You have seen one function already, `main()`, which is one for which you write the code. The C++ programming designers have already written the code for the `printf` function. At this point, you can think of `printf` as a command that outputs values to the screen, but it is actually a built-in function. Chapter 7, “Simple Input/Output” describes the `printf` function in more detail.



Put a return statement at the end of each function.

NOTE: To differentiate `printf` from regular C++ commands, parentheses are used after the name, as in `printf()`. In C++, all function names have parentheses following them. Sometimes these parentheses have something between them, and sometimes they are blank.

The last two lines in the program are shown here:

```
return 0; // ALWAYS end programs and functions with return.
}
```

The `return` command simply tells C++ that this function is finished. C++ returns control to whatever was controlling the program before it started running. In this case, because there was only one function, control is returned either to DOS or to the C++ editing environment. C++ requires a return value. Most C++ programmers return a `0` (as this program does) to the operating system. Unless you use operating-system return variables, you have little use for a return value. Until you have to be more specific, always return a `0` from `main()`.

Actually, many `return` statements are optional. C++ would know when it reached the end of the program without this statement. It is a good programming practice, however, to put a `return` statement at the end of every function, including `main()`. Because some functions require a `return` statement (if you are returning values), it is better to get in the habit of using them, rather than run the risk of leaving one out when you really need it.

You will sometimes see parentheses around the `return` value, as in:

```
return (0); // ALWAYS end programs and functions with return.
```

The parentheses are unnecessary and sometimes lead beginning C++ students into thinking that `return` is a built-in function. However, the parentheses are recommended when you want to return an expression. You read more about returning values in Chapter 19, “Function Return Values and Prototypes.”

The closing brace after the `return` does two things in this program. It signals the end of a block (begun earlier with the opening brace), which is the end of the `main()` function, and it signals the end of the program.

Review Questions

The answers to the review questions are in Appendix B, aptly named “Answers to Review Questions.”



1. What must go before each comment in a C++ program?
2. What is a variable?
3. What is a literal?



4. What are four C++ math operators?
5. What operator assigns a variable its value? (*Hint*: It is called the assignment operator.)



6. True or false: A variable can consist of only two types: integers and characters.
7. What is the operator that writes output to the screen?
8. Is the following a variable name or a string literal?

ci ty

9. What, if anything, is wrong with the following C++ statement?

```
RETURN;
```

Summary

This chapter focused on teaching you to write helpful and appropriate comments for your programs. You also learned a little about variables and literals, which hold the program's data. Without them, the term *data processing* would no longer be meaningful (there would be no data to process).

Now that you have a feel for what a C++ program looks like, it is time to begin looking at specifics of the commands. Starting with the next chapter, you begin to write your own programs. The next chapter picks up where this one left off; it takes a detailed look at literals and variables, and better describes their uses and how to choose their names.

Variables and Literals

To understand data processing with C++, you must understand how C++ creates, stores, and manipulates data. This chapter teaches you how C++ handles data by introducing the following topics:

- ◆ The concepts of variables and literals
- ◆ The types of C++ variables and literals
- ◆ Special literals
- ◆ Constant variables
- ◆ Naming and using variables
- ◆ Declaring variables
- ◆ Assigning values to variables

Garbage in, garbage out!

Now that you have seen an overview of the C++ programming language, you can begin writing C++ programs. In this chapter, you begin to write your own programs from scratch.

You learned in Chapter 3, “Your First C++ Program,” that C++ programs consist of commands and data. Datum is the heart of all C++ programs; if you do not correctly declare or use variables and literals, your data are inaccurate and your results are going to be

inaccurate as well. A computer adage says the if you put garbage in, you are going to get garbage out. This is very true. People usually blame computers for mistakes, but the computers are not always at fault. Rather, their data are often not entered properly into their programs.

This chapter spends a long time focusing on numeric variables and numeric literals. If you are not a “numbers” person, do not fret. Working with numbers is the computer’s job. You have to understand only how to tell the computer what you want it to do.

Variables

Variables have characteristics. When you decide your program needs another variable, you simply declare a new variable and C++ ensures that you get it. In C++, variable declarations can be placed anywhere in the program, as long as they are not referenced until after they are declared. To declare a variable, you must understand the possible characteristics, which follow.

- ♦ Each variable has a name.
- ♦ Each variable has a type.
- ♦ Each variable holds a value that you put there, by assigning it to that variable.

The following sections explain each of these characteristics in detail.

Naming Variables

Because you can have many variables in a single program, you must assign names to them to keep track of them. Variable names are unique, just as house addresses are unique. If two variables have the same name, C++ would not know to which you referred when you request one of them.

Variable names can be as short as a single letter or as long as 32 characters. Their names must begin with a letter of the alphabet but, after the first letter, they can contain letters, numbers, and underscore (_) characters.



TIP: Spaces are not allowed in a variable name, so use the underscore character to separate parts of the name.

The following list of variable names are all valid:

```
sal ary   aug91_sal es   i   i ndex_age   amount
```

It is traditional to use lowercase letters for C++ variable names. You do not have to follow this tradition, but you should know that uppercase letters in variable names are different from lowercase letters. For example, each of the following four variables is viewed differently by your C++ compiler.

```
sal es   Sal es   SALES   sALES
```

Be very careful with the Shift key when you type a variable name. Do not inadvertently change the case of a variable name throughout a program. If you do, C++ interprets them as distinct and separate variables.

Variables cannot have the same name as a C++ command or function. Appendix E, “Keyword and Function Reference,” shows a list of all C++ command and function names.

The following are *invalid* variable names:

```
81_sal es   Aug91+Sal es   MY AGE   pri ntf
```

Do not give variables the same name as a command or built-in function.



TIP: Although you can call a variable any name that fits the naming rules (as long as it is not being used by another variable in the program), you should always use meaningful variable names. Give your variables names that help describe the values they are holding.

For example, keeping track of total payroll in a variable called `total_payroll` is much more descriptive than using the variable name `XYZ34`. Even though both names are valid, `total_payroll` is easier to remember and you have a good idea of what the variable holds by looking at its name.

Variable Types

Variables can hold different types of data. Table 4.1 lists the different types of C++ variables. For instance, if a variable holds an integer, C++ assumes no decimal point or fractional part (the part to the right of the decimal point) exists for the variable's value. A large number of types are possible in C++. For now, the most important types you should concentrate on are `char`, `int`, and `float`. You can append the prefix `long` to make some of them hold larger values than they would otherwise hold. Using the `unsigned` prefix enables them to hold only positive numbers.

Table 4.1. Some C++ variable types.

<i>Declaration Name</i>	<i>Type</i>
<code>char</code>	Character
<code>unsigned char</code>	Unsigned character
<code>signed char</code>	Signed character (same as <code>char</code>)
<code>int</code>	Integer
<code>unsigned int</code>	Unsigned integer
<code>signed int</code>	Signed integer (same as <code>int</code>)
<code>short int</code>	Short integer
<code>unsigned short int</code>	Unsigned short integer
<code>signed short int</code>	Signed short integer (same as <code>short int</code>)
<code>long</code>	Long integer
<code>long int</code>	Long integer (same as <code>long</code>)
<code>signed long int</code>	Signed long integer (same as <code>long int</code>)
<code>unsigned long int</code>	Unsigned long integer
<code>float</code>	Floating-point
<code>double</code>	Double floating-point
<code>long double</code>	Long double floating-point

The next section more fully describes each of these types. For now, you have to concentrate on the importance of declaring them before using them.

Declaring Variables

There are two places you can declare a variable:

- ◆ Before the code that uses the variable
- ◆ Before a function name (such as before `main()` in the program)

The first of these is the most common, and is used throughout much of this book. (If you declare a variable before a function name, it is called a *global* variable. Chapter 17, “Variable Scope,” addresses the pros and cons of global variables.) To declare a variable, you must state its type, followed by its name. In the previous chapter, you saw a program that declared four variables in the following way.



Start of the `main()` function.

Declare the variables `i` and `j` as integers.

Declare the variable `c` as a character.

Declare the variable `x` as a floating-point variable.

```
main()
{
    int i, j;    // These three lines declare four variables.
    char c;
    float x;
    // The rest of program follows.
```

This declares two integer variables named `i` and `j`. You have no idea what is inside those variables, however. You generally cannot assume a variable holds zero—or any other number—until you assign it a value. The first line basically tells C++ the following:

“I am going to use two integer variables somewhere in this program. Be expecting them. I want them named `i` and `j`. When I put a value into `i` or `j`, I ensure that the value is an integer.”

Declare all variables
in a C++ program
before you use them.

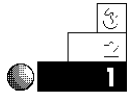
Without such a declaration, you could not assign `i` or `j` a value later. All variables must be declared before you use them. This does not necessarily hold true in other programming languages, such as BASIC, but it does for C++. You could declare each of these two variables on its own line, as in the following code:

```
main()
{
    int i;
    int j;
    // The rest of program follows.
```

You do not gain any readability by doing this, however. Most C++ programmers prefer to declare variables of the same type on the same line.

The second line in this example declares a character variable called `c`. Only single characters should be placed there. Next, a floating-point variable called `x` is declared.

Examples



1. Suppose you had to keep track of a person's first, middle, and last initials. Because an initial is obviously a character, it would be prudent to declare three character variables to hold the three initials. In C++, you could do that with the following statement:

```
main()
{
    char first, middle, last;
    // The rest of program follows.
```

This statement could go after the opening brace of `main()`. It informs the rest of the program that you require these three character variables.



2. You could declare these three variables also on three separate lines, although it does not necessarily improve readability to do so. This could be accomplished with:

```
main()
{
    char first;
    char middle;
    char last;
    // The rest of program follows.
```



3. Suppose you want to keep track of a person's age and weight. If you want to store these values as whole numbers, they would probably go in integer variables. The following statement would declare those variables:

```
main()
{
    int age, weight;
    // The rest of program follows.
```

Looking at Data Types

You might wonder why it is important to have so many variable types. After all, a number is just a number. C++ has more data types, however, than almost all other programming languages. The variable's type is critical, but choosing the type among the many offerings is not as difficult as it might first seem.

The character variable is easy to understand. A character variable can hold only a single character. You cannot put more than a single character into a character variable.



NOTE: Unlike many other programming languages, C++ does not have a string variable. Also, you cannot hold more than a single character in a C++ character variable. To store a string of characters, you must use an *aggregate* variable type that combines other fundamental types, such as an array. Chapter 5, "Character Arrays and Strings," explains this more fully.

Integers hold whole numbers. Although mathematicians might cringe at this definition, an integer is actually any number that does

not contain a decimal point. All the following expressions are integers:

45 -932 0 12 5421

Floating-point numbers contain decimal points. They are known as *real* numbers to mathematicians. Any time you have to store a salary, a temperature, or any other number that might have a fractional part (a decimal portion), you must store it in a floating-point variable. All the following expressions are floating-point numbers, and any floating-point variable can hold them:

45.12 -2344.5432 0.00 .04594

Sometimes you have to keep track of large numbers, and sometimes you have to keep track of smaller numbers. Table 4.2 shows a list of ranges that each C++ variable type can hold.



CAUTION: All true AT&T C++ programmers know that they cannot count on using the exact values in Table 4.2 on every computer that uses C++. These ranges are typical on a PC, but might be much different on another computer. Use this table only as a guide.

Table 4.2. Typical ranges that C++ variables hold.

<i>Type</i>	<i>Range*</i>
char	-128 to 127
unsigned char	0 to 255
signed char	-128 to 127
int	-32768 to 32767
unsigned int	0 to 65535
signed int	-32768 to 32767
short int	-32768 to 32767
unsigned short int	0 to 65535

<i>Type</i>	<i>Range*</i>
signed short int	-32768 to 32767
long int	-2147483648 to 2147483647
signed long int	-2147483648 to 2147483647
float	-3.4E-38 to 3.4E+38
double	-1.7E-308 to 1.7E+308
long double	-3.4E-4932 to 1.1E+4932

* Use this table only as a guide; different compilers and different computers can have different ranges.



NOTE: The floating-point ranges in Table 4.2 are shown in scientific notation. To determine the actual range, take the number before the *E* (meaning *Exponent*) and multiply it by 10 raised to the power after the plus sign. For instance, a floating-point number (type `float`) can contain a number as small as -3.4^{38} .

Notice that long integers and long doubles tend to hold larger numbers (and therefore, have a higher precision) than regular integers and regular double floating-point variables. This is due to the larger number of memory locations used by many of the C++ compilers for these data types. Again, this is usually—but not always—the case.

Do Not Over Type a Variable

If the long variable types hold larger numbers than the regular ones, you might initially want to use long variables for all your data. This would not be required in most cases, and would probably slow your program's execution.

As Appendix A, “Memory Addressing, Binary, and Hexadecimal Review,” describes, the more memory locations used by data, the larger that data can be. However, every time your computer has to access more storage for a single variable (as is usually the case for long variables), it takes the CPU much longer to access it, calculate with it, and store it.

Use the long variables only if you suspect your data might overflow the typical data type ranges. Although the ranges differ between computers, you should have an idea of whether your numbers might exceed the computer’s storage ranges. If you are working with extremely large (or extremely small and fractional) numbers, you should consider using the long variables.

Generally, all numeric variables should be signed (the default) unless you know for certain that your data contain only positive numbers. (Some values, such as age and distances, are always positive.) By making a variable an unsigned variable, you gain a little extra storage range (as explained in Appendix A, “Memory Addressing, Binary, and Hexadecimal Review”). That range of values must always be positive, however.

Obviously, you must be aware of what kinds of data your variables hold. You certainly do not always know exactly what each variable is holding, but you can have a general idea. For example, in storing a person’s age, you should realize that a long integer variable would be a waste of space, because nobody can live to an age that can’t be stored by a regular integer.

At first, it might seem strange for Table 4.2 to state that character variables can hold numeric values. In C++, integers and character variables frequently can be used interchangeably. As explained in Appendix A, “Memory Addressing, Binary, and Hexadecimal Review,” each ASCII table character has a unique number that corresponds to its location in the table. If you store a number in a character variable, C++ treats the data as if it were the ASCII character that matched that number in the table. Conversely, you can store character data in an integer variable. C++ finds that

character's ASCII number, and stores that number rather than the character. Examples that help illustrate this appear later in the chapter.

Designating Long, Unsigned, and Floating-Point Literals

When you type a number, C++ interprets its type as the smallest type that can hold that number. For example, if you print 63, C++ knows that this number fits into a signed integer memory location. It does not treat the number as a long integer, because 63 is not large enough to warrant a long integer literal size.

However, you can append a suffix character to numeric literals to override the default type. If you put an `L` at the end of an integer, C++ interprets that integer as a long integer. The number 63 is an integer literal, but the number 63L is a long integer literal.

Assign the `U` suffix to designate an unsigned integer literal. The number 63 is, by default, a signed integer literal. If you type 63U, C++ treats it as an unsigned integer. The suffix `UL` indicates an unsigned long literal.

C++ interprets all floating-point literals (numbers that contain decimal points) as double floating-point literals (double floating-point literals hold larger numbers than floating-point literals). This process ensures the maximum accuracy in such numbers. If you use the literal 6.82, C++ treats it as a double floating-point data type, even though it would fit in a regular `float`. You can append the floating-point suffix (`F`) or the long double floating-point suffix (`L`) to literals that contain decimal points to represent a floating-point literal or a long double floating-point literal.

You may rarely use these suffixes, but if you have to assign a literal value to an extended or unsigned variable, your literals might be a little more accurate if you add `U`, `L`, `UL`, or `F` (their lowercase equivalents work too) to their ends.

Assigning Values to Variables

Now that you know about the C++ variable types, you are ready to learn the specifics of assigning values to those variables. You do this with the *assignment* statement. The equal sign (=) is used for assigning values to variables. The format of the assignment statement is



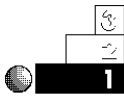
```
variable=expression;
```

The `variable` is any variable that you declared earlier. The `expression` is any variable, literal, expression, or combination that produces a resulting data type that is the same as the `variable`'s data type.



TIP: Think of the equal sign as a left-pointing arrow. Loosely, the equal sign means you want to take the number, variable, or expression on the right side of the equal sign and put it into the variable on the left side of the equal sign.

Examples



1. If you want to keep track of your current age, salary, and dependents, you could store these values in three C++ variables. You first declare the variables by deciding on correct types and good names for them. You then assign values to them. Later in the program, these values might change (for example, if the program calculates a new pay increase for you).

Good variable names include `age`, `salary`, and `dependents`. To declare these three variables, the first part of the `main()` function would look like this:

```
// Declare and store three values.
main()
{
    int age;
    float salary;
    int dependents;
```

Notice that you do not have to declare all integer variables together. The next three statements assign values to the variables.

```
age=32;
sal ary=25000. 00;
dependents=2;
// Rest of program follows.
```

This example is not very long and doesn't do much, but it illustrates the using and assigning of values to variables.



- Do not put commas in values that you assign to variables. Numeric literals should never contain commas. The following statement is invalid:

```
sal ary=25, 000. 00;
```

- You can assign variables or mathematical expressions to other variables. Suppose, earlier in a program, you stored your tax rate in a variable called `tax_rate`, then decided to use your tax rate for your spouse's rate as well. At the proper point in the program, you would code the following:

```
spouse_tax_rate = tax_rate;
```

(Adding spaces around the equal sign is acceptable to the C++ compiler, but you do not have to do so.) At this point in the program, the value in `tax_rate` is copied to a new variable named `spouse_tax_rate`. The value in `tax_rate` is still there after this line finishes. The variables were declared earlier in the program.

If your spouse's tax rate is 40 percent of yours, you can assign an expression to the spouse's variable, as in:

```
spouse_tax_rate = tax_rate * .40;
```

Any of the four mathematical symbols you learned in the previous chapter, as well as the additional ones you learn later in the book, can be part of the expression you assign to a variable.



4. If you want to assign character data to a character variable, you must enclose the character in single quotation marks. All C++ character literals must be enclosed in single quotation marks.

The following section of a program declares three variables, then assigns three initials to them. The initials are character literals because they are enclosed in single quotation marks.

```
main()
{
    char first, middle, last;
    first = 'G';
    middle = 'M';
    last = 'P';
    // Rest of program follows.
```

Because these are variables, you can reassign their values later if the program warrants it.



CAUTION: Do not mix types. C enables programmers to do this, but C++ does not. For instance, in the `middle` variable presented in the previous example, you could not have stored a floating-point literal:

```
middle = 345.43244;    // You cannot do this!
```

If you did so, `middle` would hold a strange value that would seem to be meaningless. Make sure that values you assign to variables match the variable's type. The only major exception to this occurs when you assign an integer to a character variable, or a character to an integer variable, as you learn shortly.

Literals

As with variables, there are several types of C++ literals. Remember that a literal does not change. Integer literals are whole numbers that do not contain decimal points. Floating-point literals

are numbers that contain a fractional portion (a decimal point with an optional value to the right of the decimal point).

Assigning Integer Literals

You already know that an integer is any whole number without a decimal point. C++ enables you to assign integer literals to variables, use integer literals for calculations, and print integer literals using the `cout` operator.

An octal integer literal contains a leading 0, and a hexadecimal literal contains a leading 0x.

A regular integer literal cannot begin with a leading 0. To C++, the number `012` is not the number twelve. If you precede an integer literal with a 0, C++ interprets it as an *octal* literal. An octal literal is a base-8 number. The octal numbering system is not used much in today's computer systems. The newer versions of C++ retain octal capabilities for compatibility with previous versions.

A special integer in C++ that is still greatly used today is the base-16, or *hexadecimal*, literal. Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," describes the hexadecimal numbering system. If you want to represent a hexadecimal integer literal, add the `0x` prefix to it. The following numbers are hexadecimal numbers:

`0x10` `0x2C4` `0xFFFF` `0X9`

Notice that it does not matter if you use a lowercase or uppercase letter `x` after the leading zero, or an uppercase or lowercase hexadecimal digit (for hex numbers A through F). If you write business-application programs in C++, you might think you never have the need for using hexadecimal, and you might be correct. For a complete understanding of C++ and your computer in general, however, you should become a little familiar with the fundamentals of hexadecimal numbers.

Table 4.3 shows a few integer literals represented in their regular decimal, hexadecimal, and octal notations. Each row contains the same number in all three bases.

Table 4.3. Integer literals represented in three bases.

<i>Decimal (Base 10)</i>	<i>Hexadecimal (Base 16)</i>	<i>Octal (Base 8)</i>
16	0x10	020
65536	0x10000	0100000
25	0x19	031



NOTE: Floating-point literals can begin with a leading zero, for example, 0.7. They are properly interpreted by C++. Only integers can be hexadecimal or octal literals.

Your Computer's Word Size Is Important

If you write many system programs that use hexadecimal numbers, you probably want to store those numbers in *unsigned* variables. This keeps C++ from improperly interpreting positive numbers as negative numbers.

For example, if your computer stores integers in 2-byte words (as most PCs do), the hexadecimal literal 0xFFFF represents either -1 or 65535, depending on how the sign bit is interpreted. If you declared an unsigned integer, such as

```
unsigned_int i_num = 0xFFFF;
```

C++ knows you want it to use the sign bit as data and not as the sign. If you declared the same value as a signed integer, however, as in

```
int i_num = 0xFFFF; /* The word "signed" is optional.*/
```

C++ thinks this is a negative number (-1) because the sign bit is on. (If you were to convert 0xFFFF to binary, you would get sixteen 1s.) Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," discusses these concepts in more detail.

Assigning String Literals

A string literal is always enclosed in double quotation marks.

One type of C++ literal, called the *string literal*, does not have a matching variable. A string literal is always enclosed in double quotation marks. Here are examples of string literals:

```
"C++ Programmi ng" "123" " " "4323 E. Oak Road" "x"
```

Any string of characters between double quotation marks—even a single character—is considered to be a string literal. A single space, a word, or a group of words between double quotation marks are all C++ string literals.

If the string literal contains only numeric digits, it is *not* a number; it is a string of numeric digits that you cannot use to perform mathematics. You can perform math only on numbers, not on string literals.



NOTE: A string literal is *any* character, digit, or group of characters enclosed in double quotation marks. A character literal is any character enclosed in single quotation marks.

The double quotation marks are never considered part of the string literal. The double quotation marks surround the string and simply inform your C++ compiler that the code is a string literal and not another type of literal.

It is easy to print string literals. Simply put the string literals in a `cout` statement. The following code prints a string literal to the screen:



The following code prints the string literal, C++ By Example.

```
cout << "C++ By Example";
```

Examples



1. The following program displays a simple message on-screen. No variables are needed because no datum is stored or calculated.

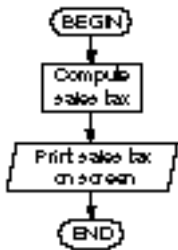
```
// Filename: C4ST1.CPP
// Display a string on-screen.

#include <iostream.h>
main()
{
    cout << "C++ programming is fun!";
    return 0;
}
```

Remember to make the last line in your C++ program (before the closing brace) a `return` statement.



2. You probably want to label the output from your programs. Do not print the value of a variable unless you also print a string literal that describes that variable. The following program computes sales tax for a sale and prints the tax. Notice a message is printed first that tells the user what the next number means.



```
// Filename: C4ST2.CPP
// Compute sales tax and display it with an appropriate message.
```

```
#include <iostream.h>
main()
{
    float sale, tax;
    float tax_rate = .08;    // Sales tax percentage

    // Determine the amount of the sale.
    sale = 22.54;

    // Compute the sales tax.
    tax = sale * tax_rate;

    // Print the results.
    cout << "The sales tax is " << tax << "\n";

    return 0;
}
```

Here is the output from the program:

```
The sales tax is 1.8032
```

You later learn how to print accurately to two decimal places to make the cents appear properly.

String-Literal Endings

An additional aspect of string literals sometimes confuses beginning C++ programmers. All string literals end with a zero. You do not see the zero, but C++ stores the zero at the end of the string in memory. Figure 4.1 shows what the string "C++ Program" looks like in memory.

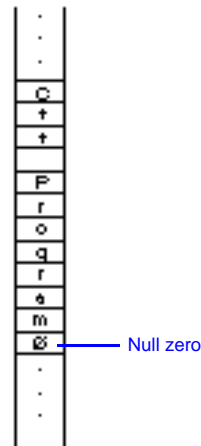


Figure 4.1. In memory, a string literal always ends with 0.

You do not have to worry about putting the zero at the end of a string literal; C++ does it for you every time it stores a string. If your program contained the string "C++ Program", for example, the compiler would recognize it as a string literal (from the double quotation marks) and store the zero at the end.

All string literals end in a null zero (also called binary zero or ASCII zero).

The zero is important to C++. It is called the *string delimiter*. Without it, C++ would not know where the string literal ended in memory. (Remember that the double quotation marks are not stored as part of the string, so C++ cannot use them to determine where the string ends.)

The string-delimiting zero is not the same as the character zero. If you look at the ASCII table in Appendix C, “ASCII Table,” you can see that the first entry, ASCII number 0, is the *null* character. (If you are unfamiliar with the ASCII table, you should read Appendix A, “Memory Addressing, Binary, and Hexadecimal Review,” for a brief description.) This string-delimiting zero is different from the character ‘0’, which has an ASCII value of 48.

As explained in Appendix A, “Memory Addressing, Binary, and Hexadecimal Review,” all memory locations in your computer actually hold bit patterns for characters. If the letter A is stored in memory, an A is not actually there; the binary bit pattern for the ASCII A (01000001) is stored there. Because the binary bit pattern for the null zero is 00000000, the string-delimiting zero is also called a *binary zero*.

To illustrate this further, Figure 4.2 shows the bit patterns for the following string literal when stored in memory: “I am 30”.

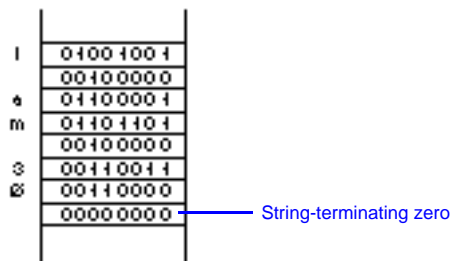


Figure 4.2. The bit pattern showing that a null zero and a character zero are different.

Figure 4.2 shows how a string is stored in your computer’s memory at the binary level. It is important for you to recognize that the character 0, inside the number 30, is not the same zero (at the bit level) as the string-terminating null zero. If it were, C++ would think this string ended after the 3, which would be incorrect.

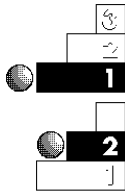
This is a fairly advanced concept, but you truly have to understand it before continuing. If you are new to computers, reviewing the material in Appendix A, “Memory Addressing, Binary, and Hexadecimal Review,” will help you understand this concept.

String Lengths

The length of a string literal does not include the null binary zero.

Many times, your program has to know the length of a string. This becomes critical when you learn how to accept string input from the keyboard. The length of a string is the number of characters up to, but not including, the delimiting null zero. Do *not* include the null character in that count, even though you know C++ adds it to the end of the string.

Examples



1. The following are all string literals:

"0" "C" "A much longer string literal"

2. The following table shows some string literals and their corresponding string lengths.

<i>String</i>	<i>Length</i>
"C"	1
"0"	21
"Hello"	5
" "	0
"30 oranges"	10

Assigning Character Literals

All C character literals should be enclosed in single quotation marks. The single quotation marks are not part of the character, but they serve to delimit the character. The following are valid C++ character literals:

'w' 'W' 'C' '7' '*' '=' '.' 'K'

C++ does not append a null zero to the end of character literals. You should know that the following are different to C++.

'R' and "R"

'R' is a single character literal. It is one character long, because *all* character literals (and variables) are one character long. "R" is a string literal because it is delimited by double quotation marks. Its length is also one, but it includes a null zero in memory so C++ knows where the string ends. Due to this difference, you cannot mix character literals and character strings. Figure 4.3 shows how these two literals are stored in memory.

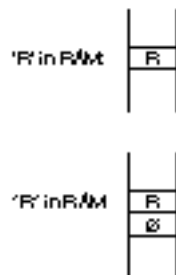


Figure 4.3. The difference in memory between 'R' as a character literal and "R" as a string literal.

All the alphabetic, numeric, and special characters on your keyboard can be character literals. Some characters, however, cannot be represented with your keyboard. They include some of the higher ASCII characters (such as the Spanish Ñ). Because you do not have keys for every character in the ASCII table, C++ enables you to represent these characters by typing their ASCII hexadecimal number inside single quotation marks.

For example, to store the Spanish Ñ in a variable, look up its hexadecimal ASCII number from Appendix C, "ASCII Table." You find that it is A5. Add the prefix \x to it and enclose it in single quotation marks, so C++ will know to use the special character. You could do that with the following code:

```
char sn=' \xA5'; // Puts the Spanish Ñ into a variable called sn.
```


This is the way to store (or print) any character from the ASCII table, even if that character does not have a key on your keyboard.

The single quotation marks still tell C++ that a *single* character is inside the quotation marks. Even though `'\xA5'` contains four characters inside the quotation marks, those four characters represent a single character, not a character string. If you were to include those four characters inside a string *literal*, C++ would treat `\xA5` as a single character in the string. The following string literal,

```
"An accented a is \xA0"
```

is a C++ string that is 18 characters, not 21 characters. C++ interprets the `\xA0` character as the `á`, just as it should.



CAUTION: If you are familiar with entering ASCII characters by typing their ASCII numbers with the Alt-keypad combination, do not do this in your C++ programs. They might work on your computer (not all C++ compilers support this), but your program might not be portable to another computer's C++ compiler.

Any character preceded by a backslash, `\`, (such as these have been) is called an *escape sequence*, or *escape character*. Table 4.4 shows some additional escape sequences that come in handy when you want to print special characters.



TIP: Include `"\n"` in a `cout` if you want to skip to the next line when printing your document.

Table 4.4. Special C++ escape-sequence characters.

<i>Escape Sequence</i>	<i>Meaning</i>
<code>\a</code>	Alarm (the terminal's bell)
<code>\b</code>	Backspace
<code>\f</code>	Form feed (for the printer)

continues

Table 4.4. Continued.

<i>Escape Sequence</i>	<i>Meaning</i>
<code>\n</code>	Newline (carriage return and line feed)
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\v</code>	Vertical tab
<code>\\</code>	Backslash (\)
<code>\?</code>	Question mark
<code>\'</code>	Single quotation mark
<code>\"</code>	Double quotation mark
<code>\000</code>	Octal number
<code>\xhh</code>	Hexadecimal number
<code>\0</code>	Null zero (or binary zero)

Math with C++ Characters

Because C++ links characters so closely with their ASCII numbers, you can perform arithmetic on character data. The following section of code,

```
char c;
c = 'T' + 5;    // Add five to the ASCII character.
```

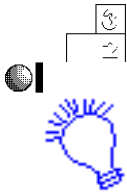
actually stores a *Y* in *c*. The ASCII value of the letter *T* is 84. Adding 5 to 84 produces 89. Because the variable *c* is not an integer variable, but is a character variable, C++ adds the ASCII character for 89, not the actual number.

Conversely, you can store character literals in integer variables. If you do, C++ stores the matching ASCII number for that character. The following section of code

```
int i = 'P';
```

does not put a letter *P* in `i` because `i` is not a character variable. C++ assigns the number 80 in the variable because 80 is the ASCII number for the letter *P*.

Examples



1. To print two names on two different lines, include the `\n` between them.

Print the name Harry; drop the cursor down to a new line and print Jerry.

```
cout << "Harry\nJerry";
```

When the program reaches this line, it prints

```
Harry
Jerry
```

You also could separate the two names by appending more of the `cout` operator, such as:

```
cout << "Harry" << "\n" << "Jerry";
```

Because the `\n` only takes one byte of storage, you can output it as a character literal by typing `'\n'` in place of the preceding `"\n"`.



2. The following short program rings the bell on your computer by assigning the `\a` escape sequence to a variable, then printing that variable.

```
// Filename: C4BELL.CPP
// Rings the bell
#include <iostream.h>
main()
{
    char bell = '\a';
    cout << bell;    // No newline needed here.
    return 0;
}
```

Constant Variables

The term *constant variable* might seem like a contradiction. After all, a constant never changes and a variable holds values that change. In C++ terminology, you can declare variables to be constants with the `const` keyword. Throughout your program, the constants act like variables; you can use a constant variable anywhere you can use a variable, but you cannot change constant variables. To declare a constant, put the keyword `const` in front of the variable declaration, for instance:

```
const int days_of_week = 7;
```

C++ offers the `const` keyword as an improvement of the `#define` preprocessor directive that C uses. Although C++ supports `#define` as well, `const` enables you to specify constant values with specific data types.

The `const` keyword is appropriate when you have data that does not change. For example, the mathematical π is a good candidate for a constant. If you accidentally attempt to store a value in a constant, C++ will let you know. Most C++ programmers choose to type their constant names in uppercase characters to distinguish them from regular variables. This is the one time when uppercase reigns in C++.



NOTE: This book reserves the name *constant* for C++ program constants declared with the `const` keyword. The term *literal* is used for numeric, character, and string data values. Some books choose to use the terms constant and literal interchangeably, but in C++, the difference can be critical.

Example



Suppose a teacher wanted to compute the area of a circle for the class. To do so, the teacher needs the value of π (mathematically, π is approximately 3.14159). Because π remains constant, it is a good candidate for a `const` keyword, as the following program shows:



Comment for the program filename and description.

Declare a constant value for π .

Declare variables for radius and area.

Compute and print the area for both radius values.

```
// Filename: C4AREAC.CPP
// Computes a circle with radius of 5 and 20.
#include <iostream.h>
main()
{
    const float PI=3.14159;
    float radius = 5;
    float area;

    area = radius * radius * PI; // Circle area calculation
    cout << "The area is " << area << " with a radius of 5.\n";

    radius = 20; // Compute area with new radius.
    area = radius * radius * PI;
    cout << "The area is " << area << " with a radius of 20.\n";

    return 0;
}
```

Review Questions

The answers to the review questions are in Appendix B.

1. Which of the following variable names are valid?

my_name 89_sal es sal es_89 a-sal ary

2. Which of the following literals are characters, strings, integers, and floating-point literals?

0 -12.0 "2.0" "X" 'X' 65.4 -708 '0'



3. How many variables do the following statements declare, and what are their types?

```
int i, j, k;
char c, d, e;
float x=65.43;
```



4. With what do all string literals end?



5. True or false: An unsigned variable can hold a larger value than a signed variable.

6. How many characters of storage does the following literal take?

```
'\x41'
```

7. How is the following string stored at the bit level?

```
"Order 10 of them."
```

8. How is the following string (called a *null string*) stored at the bit level? (*Hint*: The length is zero, but there is still a terminating character.)

```
""
```

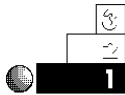
9. What is wrong with the following program?

```
#include <iostream.h>
main()
{
    const int age=35;
    cout << age << "\n";

    age = 52;
    cout << age << "\n";
    return 0;
}
```

Review Exercises

Now that you have learned some basic C++ concepts, the remainder of the book will include this section of review exercises so you can practice your programming skills.



1. Write the C++ code to store three variables: your weight (you can fib), height in feet, and shoe size. Declare the variables, then assign their values in the body of your program.



2. Rewrite your program from Exercise 1, adding proper `cout` statements to print the values to the screen. Use appropriate messages (by printing string literals) to describe the numbers that are printed.



3. Write a program that stores a value and prints each type of variable you learned in this chapter.
4. Write a program that stores a value into every type of variable C++ allows. You must declare each variable at the beginning of your program. Give them values and print them.

Summary

A firm grasp of C++'s fundamentals is critical to a better understanding of the more detailed material that follows. This is one of the last general-topic chapters in the book. You learned about variable types, literal types, how to name variables, how to assign variable values, and how to declare constants. These issues are *critical* to understanding the remaining concepts in C++.

This chapter taught you how to store almost every type of literal into variables. There is no string variable, so you cannot store string literals in string variables (as you can in other programming languages). However, you can “fool” C++ into *thinking* it has a string variable by using a character array to hold strings. You learn this important concept in the next chapter.

Character Arrays and Strings

Even though C++ has no string variables, you can act as if C++ has them by using character arrays. The concept of arrays might be new to you, but this chapter explains how easy they are to declare and use. After you declare these arrays, they can hold character strings—just as if they were real string variables. This chapter includes

- ◆ Character arrays
- ◆ Comparison of character arrays and strings
- ◆ Examples of character arrays and strings

After you master this chapter, you are on your way to being able to manipulate almost every type of variable C++ offers. Manipulating characters and words is one feature that separates your computer from a powerful calculator; this capability gives computers true data-processing capabilities.

Character Arrays

A string literal can be stored in an array of characters.

Almost every type of data in C++ has a variable, but there is no variable for holding character strings. The authors of C++ realized that you need some way to store strings in variables, but instead of storing them in a string variable (as some languages such as BASIC or Pascal do) you must store them in an *array* of characters.

If you have never programmed before, an array might be new to you. An array is a list (sometimes called a *table*) of variables, and most programming languages allow the use of such lists. Suppose you had to keep track of the sales records of 100 salespeople. You could make up 100 variable names and assign a different salesperson's sales record to each one.

All those different variable names, however, are difficult to track. If you were to put them in an array of floating-point variables, you would have to keep track of only a single name (the array name) and reference each of the 100 values by a numeric subscript.

The last few chapters of this book cover array processing in more detail. However, to work with character string data in your early programs, you have to become familiar with the concept of *character arrays*.

Because a string is simply a list of one or more characters, a character array is the perfect place to hold strings of information. Suppose you want to keep track of a person's full name, age, and salary in variables. The age and salary are easy because there are variable types that can hold such data. The following code declares those two variables:

```
int age;  
float salary;
```

You have no string variable to hold the name, but you can create an appropriate array of characters (which is actually one or more character variables in a row in memory) with the following declaration:

```
char name[15];
```

This reserves a character array. An array declaration always includes brackets ([]) that declare the space for the array. This array is 15 characters long. The array name is `name`. You also can assign a

EXAMPLE

value to the character array at the time you declare the array. The following declaration statement not only declares the character array, but also assigns the name “Michael Jones” at the same time:



Declare the character array called name as 15 characters long, and assign Michael Jones to the array.

```
char name[15]="Michael Jones";
```

Figure 5.1 shows what this array looks like in memory. Each of the 15 boxes of the array is called an *element*. Notice the null zero (the string-terminating character) at the end of the string. Notice also that the last character of the array contains no data. You filled only the first 14 elements of the array with the data and the data’s null zero. The 15th element actually has a value in it—but whatever follows the string’s null zero is not a concern.

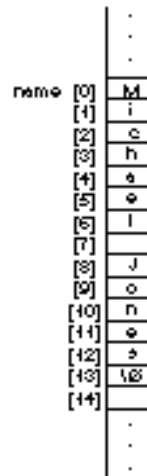


Figure 5.1. A character array after being declared and assigned a string value.

You can access individual elements in an array, or you can access the array as a whole. This is the primary advantage of an array over the use of many differently named variables. You can assign values to the individual array elements by putting the elements’ location, called a *subscript*, in brackets, as follows:

```
name[3]='k';
```

This overwrites the `h` in the name `Mi chael` with a `k`. The string now looks like the one in Figure 5.2.



Figure 5.2. The array contents (see Figure 5.1) after changing one of the elements.

All array subscripts begin at 0.

All array subscripts start at zero. Therefore, to overwrite the first element, you must use 0 as the subscript. Assigning `name[3]` (as is done in Figure 5.2) changes the value of the fourth element in the array.

You can print the entire string—or, more accurately, the entire array—with a single `cout` statement, as follows:

```
cout << name;
```

Notice when you print an array, you do not include brackets after the array name. You must be sure to reserve enough characters in the array to hold the entire string. The following line,

```
char name[5]="Mi chael Jones";
```

is incorrect because it reserves only five characters for the array, whereas the name and its null zero require 14 characters. However, C++ does give you an error message for this mistake (`illegal initialization`).



CAUTION: Always reserve enough array elements to hold the string, plus its null-terminating character. It is easy to forget the null character, but don't do it!

If your string contains 13 characters, it also must have a 14th for the null zero or it will never be treated like a string. To help eliminate this error, C++ gives you a shortcut. The following two character array statements are the same:

```
char horse[9]="Stallion";
```

and

```
char horse[]="Stallion";
```

If you assign a value to a character array at the same time you declare the array, C++ counts the string's length, adds one for the null zero, and reserves the array space for you.

If you do not assign a value to an array at the time it is declared, you cannot declare it with empty brackets. The following statement,

```
char people[];
```

does not reserve any space for the array called `people`. Because you did not assign a value to the array when you declared it, C++ assumes this array contains zero elements. Therefore, you have no room to put values in this array later. Most compilers generate an error if you attempt this.

Character Arrays Versus Strings

In the previous section, you saw how to put a string in a character array. Strings can exist in C++ only as string *literals*, or as stored information in character arrays. At this point, you have only to understand that strings must be stored in character arrays. As you read through this book and become more familiar with arrays and strings, however, you should become more comfortable with their use.



NOTE: Strings must be stored in character arrays, but not all character arrays contain strings.

Look at the two arrays shown in Figure 5.3. The first one, called `cara1`, is a character array, but it does not contain a string. Rather than a string, it contains a list of several characters. The second array, called `cara2`, contains a string because it has a null zero at its end.

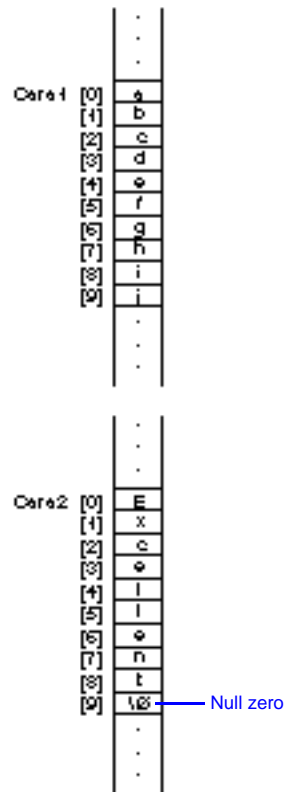


Figure 5.3. Two character arrays: `Car1` contains characters, and `Car2` contains a character string.

You could initialize these arrays with the following assignment statements.

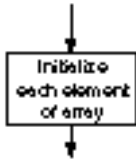


Declare the array `cara1` with 10 individual characters.

Declare the array `cara2` with the character string "Excel l ent".

```
char cara1[10]={'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
               'j'};
char cara2[10]="Excel l ent";
```

If you want to put only individual characters in an array, you must enclose the list of characters in braces, as shown. You could initialize `cara1` later in the program, using assignment statements, as the following code section does.



```
char cara1[10];
cara1[0]='a';
cara1[1]='b';
cara1[2]='c';
cara1[3]='d';
cara1[4]='e';
cara1[5]='f';
cara1[6]='g';
cara1[7]='h';
cara1[8]='i';
cara1[9]='j'; // Last element possible with subscript of nine.
```

Because the `cara1` character array does not contain a null zero, it does not contain a string of characters. It does contain characters that can be stored in the array—and used individually—but they cannot be treated in a program as if they were a string.



CAUTION: You cannot assign string values to character arrays in a regular assignment statement, except when you first declare the character arrays.

Because a character array is not a string variable (it can be used only to hold a string), it cannot go on the left side of an equal (=) sign. The program that follows is invalid:

```

#include <iostream.h>
main()
{
    char petname[20];    // Reserve space for the pet's name.
    petname = "Al fal fa"; // INVALID!
    cout << petname;    // The program will never get here.
    return;
}

```

Because the pet's name was not assigned *at the time the character array was declared*, it cannot be assigned a value later. The following is allowed, however, because you can assign values individually to a character array:

```

#include <iostream.h>
main()
{
    char petname[20]; // Reserve space for the pet's name.
    petname[0]='A'; // Assign values one element at a time.
    petname[1]='l';
    petname[2]='f';
    petname[3]='a';
    petname[4]='l';
    petname[5]='f';
    petname[6]='a';
    petname[7]='\0'; // Needed to ensure this is a string!
    cout <<petname; // Now the pet's name prints properly.
    return;
}

```

The `petname` character array now holds a string because the last character is a null zero. How long is the string in `petname`? It is seven characters long because the length of a string never includes the null zero.

You cannot assign more than 20 characters to this array because its reserved space is only 20 characters. However, you can store any string of 19 (leaving one for the null zero) or fewer characters to the array. If you assign the "Al fal fa" string in the array as shown, and then assign a null zero to `petname[3]` as in:

```
petname[3]='\0';
```


the string in `petname` is now only three characters long. You have, in effect, shortened the string. There are still 20 characters reserved for `petname`, but the data inside it is the string "Al f" ending with a null zero.

There are many other ways to assign a value to a string. You can use the `strcpy()` function, for example. This is a built-in function that enables you to copy a string literal in a string. To copy the "Al fal fa" pet name into the `petname` array, you type:

```
strcpy(petname, "Al fal fa"); // Copies Al fal fa i into the array.
```

The `strcpy()` function puts string literals in string arrays.

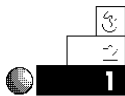
The `strcpy()` ("string copy") function assumes that the first value in the parentheses is a character array name, and that the second value is a valid string literal or another character array that holds a string. You must be sure that the first character array in the parentheses is long enough (in number of reserved elements) to hold whatever string you copy into it.



NOTE: Place an `#include <string.h>` line before the `main()` function in programs that use `strcpy()` or any other built-in string functions mentioned in this book. Your compiler supplies the `string.h` file to help the `strcpy()` function work properly. The `#include` files such as `iostream.h` and `string.h` will be further explained as you progress through this book.

Other methods of initializing arrays are explored throughout the rest of this book.

Examples



1. Suppose you want to keep track of your aunt's name in a program so you can print it. If your aunt's name is Ruth Ann Cooper, you have to reserve at least 16 elements—15 to hold the name and one to hold the null character. The following statement properly reserves a character array to hold her name:

```
char aunt_name[16];
```

2. If you want to put your aunt's name in the array at the same time you reserve the array space, you could do it like this:

```
char aunt_name[16]="Ruth Ann Cooper";
```

You could also leave out the array size and allow C++ to count the number needed:

```
char aunt_name[]="Ruth Ann Cooper";
```



3. Suppose you want to keep track of the names of three friends. The longest name is 20 characters (including the null zero). You simply have to reserve enough character-array space to hold each friend's name. The following code does the trick:

```
char friend1[20];
char friend2[20];
char friend3[20];
```

These array declarations should appear toward the top of the block, along with any integer, floating-point, or character variables you have to declare.

4. The next example asks the user for a first and last name. Use the `cin` operator (the opposite of `cout`) to retrieve data from the keyboard. Chapter 7, "Simple I/O," more fully explains the `cout` and `cin` operators. The program then prints the user's initials on-screen by printing the first character of each name in the array. The program must print each array's `0` subscript because the first subscript of any array begins at `0`, not `1`.

```
// Filename: C5INIT.CPP
// Print the user's initials.
#include <iostream.h>
main()
{
    char first[20];    // Holds the first name
    char last[20];    // Holds the last name

    cout << "What is your first name? \n";
    cin >> first;
```

```

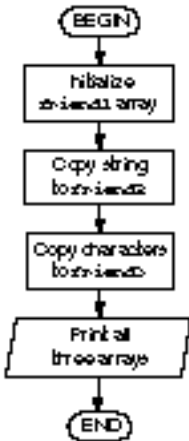
cout << "What is your last name? \n";
cin >> last;

// Print the initials
cout << "Your initials are " << first[0] << " "
    << last[0];
return 0;
}

```



5. The following program takes your three friends' character arrays and assigns them string values by using the three methods shown in this chapter. Notice the extra `#include` file used with the string function `strcpy()`.



```

// Filename: C5STR.CPP
// Store and initialize three character arrays for three
friends.

```

```

#include <iostream.h>
#include <string.h>
main()
{
    // Declare all arrays and initialize the first one.
    char friend1[20]="Jackie Paul Johnson";
    char friend2[20];
    char friend3[20];

    // Use a function to initialize the second array.
    strcpy(friend2, "Julie L. Roberts");

```

```

    friend3[0]='A'; // Initialize the last,
    friend3[1]='d'; // an element at a time.
    friend3[2]='a';
    friend3[3]='m';
    friend3[4]=' ';
    friend3[5]='G';
    friend3[6]='.';
    friend3[7]=' ';
    friend3[8]='S';
    friend3[9]='m';
    friend3[10]='i';

```

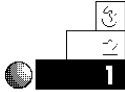
```
friend3[11]='t';
friend3[12]='h';
friend3[13]='\0';

// Print all three names.
cout << friend1 << "\n";
cout << friend2 << "\n";
cout << friend3 << "\n";
return 0;
}
```

The last method of initializing a character array with a string—one element at a time—is not used as often as the other methods.

Review Questions

The answers to the review questions are in Appendix B.



1. How would you declare a character array called `my_name` that holds the following string literal?

```
"This is C++"
```

2. How long is the string in Question 1?
3. How many bytes of storage does the string in Question 1 take?



4. With what do all string literals end?
5. How many variables do the following statements declare, and what are their types?

```
char name[25];
char address[25];
```

6. True or false: The following statement assigns a string literal to a character array.

```
myname[]="Kim Langston";
```



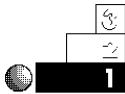
7. True or false: The following declaration puts a string in the character array called `ci ty`.

```
char ci ty[]={ 'M', 'i', 'a', 'm', 'i', '\0' };
```

8. True or false: The following declaration puts a string in the character array called `ci ty`.

```
char ci ty[]={ 'M', 'i', 'a', 'm', 'i' };
```

Review Exercises



1. Write the C++ code to store your weight, height (in feet), shoe size, and name with four variables. Declare the variables, then assign their values in the body of your program.



2. Rewrite the program in Exercise 1, adding proper `printf()` statements to print the values. Use appropriate messages (by printing string literals) to describe the printed values.



3. Write a program to store and print the names of your two favorite television programs. Store these programs in two character arrays. Initialize one of the strings (assign it the first program's name) at the time you declare the array. Initialize the second value in the body of the program with the `strcpy()` function.
4. Write a program that puts 10 different initials in 10 elements of a single character array. Do not store a null zero. Print the list backward, one initial on each line.

Summary

This has been a short, but powerful chapter. You learned about character arrays that hold strings. Even though C++ has no string variables, character arrays can hold string literals. After you put a string in a character array, you can print or manipulate it as if it were a string.

Starting with the next chapter, you begin to hone the C++ skills you are building. Chapter 6, “Preprocessor Directives,” introduces preprocessor directives, which are not actually part of the C++ language but help you work with your source code before your program is compiled.

Preprocessor Directives

As you might recall from Chapter 2, “What Is a Program?,” the C++ compiler routes your programs through a *preprocessor* before it compiles them. The preprocessor can be called a “pre-compiler” because it preprocesses and prepares your source code for compiling before your compiler receives it.

Because this *preprocess* is so important to C++, you should familiarize yourself with it before learning more specialized commands in the language. Regular C++ commands do not affect the preprocessor. You must supply special non-C++ commands, called *preprocessor directives*, to control the preprocessor. These directives enable you, for example, to modify your source code before the code reaches the compiler. To teach you about the C++ preprocessor, this chapter

- ◆ Defines preprocessor directives
- ◆ Introduces the `#include` preprocessor directive
- ◆ Introduces the `#define` preprocessor directive
- ◆ Provides examples of both

Almost every proper C++ program contains preprocessor directives. This chapter teaches you the two most common: `#include` and `#define`.

Understanding Preprocessor Directives

Preprocessor directives are commands that you supply to the preprocessor. All preprocessor directives begin with a pound sign (#). Never put a semicolon at the end of preprocessor directives, because they are preprocessor commands and not C++ commands. Preprocessor directives typically begin in the first column of your source program. They can begin in any column, of course, but you should try to be consistent with the standard practice and start them in the first column wherever they appear. Figure 6.1 illustrates a program that contains three preprocessor directives.

Preprocessor
directives

```
// Filename: C6PRE.CPP
// C++ program that demonstrates preprocessor directives.

#include <iostream.h>
#define AGE 28
#define MESSAGE "Hello, world"

main()
{
    int i = 10, age;    // i is assigned a value at declaration
                     // age is still UNDEFINED

    age = 5;          // Defines the variable, age, as five.

    i = i * AGE;      // AGE is not the same as the variable, age.

    cout << i << " " << age << " " << AGE << "\n"; // 280 5 28
    cout << MESSAGE; // Prints "Hello world".

    return 0;
}
```

Figure 6.1. Program containing three preprocessor directives.

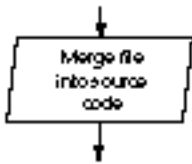
Preprocessor directives temporarily change your source code.

Preprocessor directives cause your C++ preprocessor to change your source code, but these changes last only as long as the compilation. When you look at your source code again, the preprocessor is finished with your file and its changes are no longer in the file. Your preprocessor does not in any way compile your program or change your actual C++ commands. This concept confuses some beginning C++ students, but just remember that your program has yet to be compiled when your preprocessor directives execute.

It has been said that a preprocessor is nothing more than a text-editor on your program. This analogy holds true throughout this chapter.

The #include Directive

The `#include` preprocessor directive merges a disk file into your source program. Remember that a preprocessor directive does nothing more than a word processing command does to your program; word processors also are capable of file merging. The format of the `#include` preprocessor directive follows:



```
#include <filename>
```

or

```
#include "filename"
```

In the `#include` directive, the `filename` must be an ASCII text file (as your source file must be) located somewhere on a disk. To better illustrate this rule, it might help to leave C++ for just a moment. The following example shows the contents of two files on disk. One is called OUTSIDE and the other is called INSIDE.

These are the contents of the OUTSIDE file:

```
Now is the time for all good men
```

```
#include <INSIDE>
```

```
to come to the aid of their country.
```

The INSIDE file contains the following:

```
A quick brown fox jumped
over the lazy dog.
```

Assume you can run the OUTSIDE file through the C++ preprocessor, which finds the `#include` directive and replaces it with the entire file called INSIDE. In other words, the C++ preprocessor directive merges the INSIDE file into the OUTSIDE file—at the `#include` location—and OUTSIDE expands to include the merged text. After the preprocessing ends, OUTSIDE looks like this:

```
Now is the time for all good men

A quick brown fox jumped
over the lazy dog.

to come to the aid of their country.
```

The INSIDE file remains on disk in its original form. Only the file containing the `#include` directive is changed. This change is only temporary; that is, OUTSIDE is expanded by the included file only for as long as it takes to compile the program.

A few real-life examples might help, because the OUTSIDE and INSIDE files are not C++ programs. You might want to include a file containing common code that you frequently use. Suppose you print your name and address quite often. You can type the following few lines of code in every program that prints your name and address:

```
cout << "Kelly Jane Peterson\n";
cout << "Apartment #217\n";
cout << "4323 East Skelly Drive\n";
cout << "New York, New York\n";
cout << "          10012\n";
```

Instead of having to retype the same five lines again and again, you type them once and save them in a file called MYADD.C. From then on, you only have to type the single line:

```
#include <myadd.c>
```

This not only saves typing, but it also maintains consistency and accuracy. (Sometimes this kind of repeated text is known as a *boilerplate*.)

You usually can use angled brackets, `<>`, or double quotation marks, `" "`, around the included filename with the same results. The angled brackets tell the preprocessor to look for the *include* file in a default include directory, set up by your compiler. The double quotation marks tell the preprocessor first to look for the include file in the directory where the source code is stored, and then, to look for it in the system's include directory.

Most of the time, you do see angled brackets around the included filename. If you want to include sections of code in other programs, be sure to store that code in the system's include directory (if you use angled brackets).

Even though `#include` works well for inserted source code, there are other ways to include common source code that are more efficient. You learn about one technique, called writing *external functions*, in Chapter 16, "Writing C++ Functions."

This source code `#include` example serves well to explain what the `#include` preprocessor directive does. Despite this fact, `#include` seldom is used to include source code text, but is more often used to include special system files called *header* files. These system files help C++ interpret the many built-in functions that you use. Your C++ compiler comes with its own header files. When you (or your system administrator) installed your C++ compiler, these header files were automatically stored on your hard drive in the system's include directory. Their filenames always end in `.h` to differentiate them from regular C++ source code.

The most common header file is named `iostream.h`. This file gives your C++ compiler needed information about the built-in `cout` and `cin` operators, as well as other useful built-in routines that perform input and output. The name "*iostream.h*" stands for *input/output stream header*.

At this point, you don't have to understand the `iostream.h` file. You only have to place this file before `main()` in every program you write. It is rare that a C++ program does not need the `iostream.h` file. Even when the file is not needed, including it does no harm. Your programs can work without `iostream.h` as long as they do not use

The `#include` directive is most often used for system header files.

an input or output operator defined there. Nevertheless, your programs are more accurate and hidden errors come to the surface much faster if you include this file.

Throughout this book, whenever a new built-in function is described, the function’s matching header file is included. Because almost every C++ program you write includes a `cout` to print to the screen, almost every program contains the following line:



Include the built-in C++ header file called `iostream.h`.

```
#include <iostream.h>
```

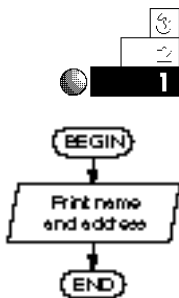
In the last chapter, you saw the `strcpy()` function. Its header file is called `string.h`. Therefore, if you write a program that contains `strcpy()`, include its matching header file at the same time you include `<iostream.h>`. These appear on separate lines, such as:

```
#include <iostream.h>
#include <string.h>
```

The order of your include files does not matter as long as you include the files before the functions that need them. Most C++ programmers include all their needed header files before `main()`.

These header files are simply text files. If you like, find a header file such as `stdio.h` on your hard drive and look at it. The file might seem complex at this point, but there is nothing “hidden” about it. Don’t change the header file in any way while looking at it. If you do, you might have to reload your compiler to restore the file.

Examples



1. The following program is short. It includes the name-and-address printing routine described earlier. After printing the name and address, it ends.

```
// Filename: C61NC1.CPP
// Illustrates the #include preprocessor directives.
#include <iostream.h>
```

```
main()
{
#include "myadd.c"
return 0;
}
```

The double quotation marks are used because the file called MYADD.C is stored in the same directory as the source file. Remember that if you type this program into your computer (after typing and saving the MYADD.C file) and then compile your program, the MYADD.C file is included only as long as it takes to compile the program. Your compiler does not see this file. Your compiler acts as if you have typed the following:

```
// Filename: C61NCL1.CPP
// Illustrates the #include preprocessor directive.
#include <iostream.h>
main()
{
cout<<"Kelly Jane Peterson\n";
cout<<"Apartment #217\n";
cout<<"4323 East Skelly Drive\n";
cout<<"New York, New York\n";
cout<<"          10012\n";
return 0;
}
```

This explains what is meant by a preprocessor: The changes are made to your source code before it's compiled. Your original source code is restored as soon as the compile is finished. When you look at your program again, it appears as originally typed, with the `#include` statement.



- The following program copies a message into a character array and prints it to the screen. Because the `cout` and `strcpy()` built-in functions are used, both of their header files are included.

```

// Filename: C61NCL3.CPP
// Uses two header files.

#include <iostream.h>
#include <string.h>

main()
{
    char message[20];
    strcpy(message, "This is fun!");
    cout << message;
    return 0;
}

```

The #define Directive

The `#define` preprocessor directive is used in C++ programming, although not nearly as frequently as it is in C. Due to the `const` keyword (in C++) that enables you to define variables as constants, `#define` is not used as much in C++. Nevertheless, `#define` is useful for compatibility to C programs you are converting to C++. The `#define` directive might seem strange at first, but it is similar to a search-and-replace command on a word processor. The format of `#define` follows:

```
#define ARGUMENT1 argument2
```

where `ARGUMENT1` is a single word containing no spaces. Use the same naming rules for the `#define` statement's first argument as for variables (see Chapter 4, "Variables and Literals"). For the first argument, it is traditional to use uppercase letters—one of the only uses of uppercase in the entire C++ language. At least one space separates `ARGUMENT1` from `argument2`. The `argument2` can be any character, word, or phrase; it also can contain spaces or anything else you can type on the keyboard. Because `#define` is a preprocessor directive and not a C++ command, do not put a semicolon at the end of its expression.

The `#define` preprocessor directive replaces the occurrence of `ARGUMENT1` everywhere in your program with the contents of

The `#define` directive replaces every occurrence of a first argument with a second argument.

argument2. In most cases, the `#define` directive should go before `main()` (along with any `#include` directives). Look at the following `#define` directive:



Define the `AGELIMIT` literal to 21.

```
#define AGELIMIT 21
```

If your program includes one or more occurrences of the term `AGELIMIT`, the preprocessor replaces every one of them with the number 21. The compiler then reacts as if you actually had typed 21 rather than `AGELIMIT`, because the preprocessor changes all occurrences of `AGELIMIT` to 21 before your compiler reads the source code. But, again, the change is only temporary. After your program is compiled, you see it as you originally typed it, with `#define` and `AGELIMIT` still intact.

`AGELIMIT` is not a variable, because variables are declared and assigned values only at the time when your program is compiled and run. The preprocessor changes your source file before the time it is compiled.

You might wonder why you would ever have to go to this much trouble. If you want 21 everywhere `AGELIMIT` occurs, you could type 21 to begin with! But the advantage of using `#define` rather than literals is that if the age limit ever changes (perhaps to 18), you have to change only one line in the program, not every single occurrence of the literal 21.

Because `#define` enables you easily to define and change literals, the replaced arguments of the `#define` directive are sometimes called *defined literals*. (C programmers say that `#define` “defines constants,” but C++ programmers rarely use the word “constant” unless they are discussing the use of `const`.) You can define any type of literal, including string literals. The following program contains a defined string literal that replaces a string in two places.

```
// Filename: C6DEF1.CPP
// Defines a string literal and uses it twice.

#include <iostream.h>
#define MYNAME "Phil Ward"

main()
```

The `#define` directive creates defined literals.

```

{
    char name[]=MYNAME;
    cout << "My name is " << name << "\n";    // Prints the array.
    cout << "My name is " << MYNAME << "\n"; // Prints the
                                                // defined literal.

    return 0;
}

```

The first argument of `#define` is in uppercase to distinguish it from variable names in the program. Variables are usually typed in lowercase. Although your preprocessor and compiler will not confuse the two, other users who look at your program can more quickly scan through and tell which items are defined literals and which are not. They will know when they see an uppercase word (if you follow the recommended standard for this first `#define` argument) to look at the top of the program for its actual defined value.

The fact that defined literals are not variables is even more clear in the following program. This program prints five values. Try to guess what those five values are before you look at the answer following the program.

```

// Filename: C6DEF2.CPP
// Illustrates that #define literals are not variables.

#include <iostream.h>

#define X1 b+c
#define X2 X1 + X1
#define X3 X2 * c + X1 - d
#define X4 2 * X1 + 3 * X2 + 4 * X3

main()
{
    int b = 2;    // Declares and initializes four variables.
    int c = 3;
    int d = 4;
    int e = X4;
    // Prints the values.
    cout << e << ", " << X1 << ", " << X2;
    cout << ", " << X3 << ", " << X4 << "\n";
    return 0;
}

```


The output from this program is

```
44 5 10 17 44
```

If you treated `x1`, `x2`, `x3`, and `x4` as variables, you would not receive the correct answers. `x1` through `x4` are not variables; they are defined literals. Before your program is compiled, the preprocessor reads the first line and changes every occurrence of `x1` to `b+c`. This occurs before the next `#define` is processed. Therefore, after the first `#define`, the source code looks like this:

```
// Filename: C6DEF2.CPP
// Illustrates that #define literals are not variables.

#include <iostream.h>

#define X2 b+c + b+c
#define X3 X2 * c + b+c - d
#define X4 2 * b+c + 3 * X2 + 4 * X3

main()
{
    int b=2;      // Declares and initializes four variables.
    int c=3;
    int d=4;
    int e=X4;

    // Prints the values.
    cout << e << ", " << b+c << ", " << X2;
    cout << ", " << X3 << ", " << X4 << "\n";
    return 0;
}
```

After the first `#define` finishes, the second one takes over and changes every occurrence of `x2` to `b+c + b+c`. Your source code at that point becomes:

```
// Filename: C6DEF2.CPP
// Illustrates that #define literals are not variables.

#include <iostream.h>
```

```

#define X3 b+c + b+c * c + b+c - d
#define X4 2 * b+c + 3 * b+c + b+c + 4 * X3

main()
{
    int b=2;    // Declares and initializes four variables.
    int c=3;
    int d=4;
    int e=X4;

    // Prints the values.
    cout << e << ", " << b+c << ", " << b+c + b+c;
    cout << ", " << X3 << ", " << X4 << "\n";
    return 0;
}

```

After the second `#define` finishes, the third one takes over and changes every occurrence of `X3` to `b+c + b+c * c + b+c - d`. Your source code then becomes:

```

// Filename: C6DEF2.CPP
// Illustrates that #define literals are not variables.

#include <iostream.h>

#define X4 2 * b+c + 3 * b+c + b+c + 4 * b+c + b+c * c + b+c - d

main()
{
    int b=2;    // Declares and initializes four variables.
    int c=3;
    int d=4;
    int e=X4;

    // Prints the values.
    cout << e << ", " << b+c << ", " << b+c + b+c;
    cout << ", " << b+c + b+c * c + b+c - d
        << ", " << X4 << "\n";
    return 0;
}

```

The source code is growing rapidly! After the third `#define` finishes, the fourth and last one takes over and changes every occurrence of `X4` to `2 * b+c + 3 * b+c + b+c + 4 * b+c + b+c * c + b+c - d`. Your source code at this last point becomes:

```
// Filename: C6DEF2.CPP
// Illustrates that #define literals are not variables.

#include <iostream.h>

main()
{
    int b=2;      // Declares and initializes four variables.
    int c=3;
    int d=4;
    int e=2 * b+c + 3 * b+c + b+c + 4 * b+c + b+c * c + b+c - d;

    // Prints the values.
    cout << e << ", " << b+c << ", " << b+c + b+c;
    cout << ", " << b+c + b+c * c + b+c - d
         << ", " << 2 * b+c + 3 * b+c + b+c + 4 * b+c +
         b+c * c + b+c - d << "\n";
    return 0;
}
```

This is what your compiler actually reads. You did not type this complete listing; you typed the original listing (shown first). The preprocessor expanded your source code into this longer form, just as if you had typed it this way.

This is an extreme example, but it serves to illustrate how `#define` works on your source code and doesn't define any variables. The `#define` behaves like a word processor's search-and-replace command. Due to `#define`'s behavior, you can even rewrite the C++ language!

If you are used to BASIC, you might be more comfortable typing `PRINT` rather than C++'s `cout` when you want to print on-screen. If so, the following `#define` statement,

```
#define PRINT cout
```

enables you to print in C++ with these statements:

```
PRINT << "This is a new printing technique\n";
PRINT << "I could have used cout instead.\n";
```

This works because by the time your compiler reads the program, it reads only the following:

```
cout << "This is a new printing technique\n";
cout << "I could have used cout instead.\n";
```

In the next chapter, “Simple Input/Output,” you learn about two functions sometimes used for input and output called `printf()` and `scanf()`. You can just as easily redefine function names using `#define` as you did with `cout`.

Also, remember that you cannot replace a defined literal if it resides in another string literal. For example, you cannot use the following `#define` statement:

```
#define AGE
```

to replace information in this `cout`:

```
cout << "AGE";
```

because `AGE` is a string literal, and it prints literally just as it appears inside the double quotation marks. The preprocessor can replace only defined literals that do not appear in quotation marks.

Do Not Overdo `#define`

Many early C programmers enjoyed redefining parts of the language to suit whatever they were used to in another language. The `cout` to `PRINT` example was only one example of this. You can redefine virtually any C++ statement or function to “look” any way you like.

There is a danger to this, however, so be wary of using `#define` for this purpose. Your redefining the language becomes confusing to others who modify your program later. Also, as you become more familiar with C++, you will naturally use the true

C++ language more and more. When you are comfortable with C++, older programs that you redefined will be confusing—even to you!

If you are programming in C++, use the language conventions that C++ provides. Shy away from trying to redefine commands in the language. Think of the `#define` directive as a way to define numeric and string literals. If those literals ever change, you have to change only one line in your program. “Just say no” to any temptation to redefine commands and built-in functions. Better yet, modify any older C code that uses `#define`, and replace the `#define` preprocessor directive with the more useful `const` command.

Examples



1. Suppose you want to keep track of your company’s target sales amount of \$55,000.00. That target amount has not changed for the previous two years. Because it probably will not change soon (sales are flat), you decide to start using a defined literal to represent this target amount. Then, if target sales do change, you just have to change the amount on the `#define` line to:

```
#define TARGETSALES 55000.00
```

which defines a floating-point literal. You can then assign `TARGETSALES` to floating-point variables and print its value, just as if you had typed `55000.00` throughout your program, as these lines show:

```
amt = TARGETSALES  
cout << TARGETSALES;
```



2. If you find yourself defining the same literals in many programs, file the literals on disk and include them. Then, you don’t have to type your defined literals at the beginning

of every program. If you store these literals in a file called MYDEFS.C in your program's directory, you can include the file with the following `#include` statement:

```
#include "mydefs.c"
```

(To use angled brackets, you have to store the file in your system's include directory.)



3. Defined literals are appropriate for array sizes. For example, suppose you declare an array for a customer's name. When you write the program, you know you don't have a customer whose name is longer than 22 characters (including the null). Therefore, you can do this:

```
#define CNMLENGTH 22
```

When you define the array, you can use this:

```
char cust_name[CNMLENGTH]
```

Other statements that need the array size also can use CNMLENGTH.



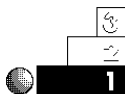
4. Many C++ programmers define a list of error messages. Once they define the messages with an easy-to-remember name, they can print those literals if an error occurs and still maintain consistency in their programs. The following error messages (or a similar form) often appear at the beginning of C++ programs.



```
#define DISKERR "Your disk drive seems not to be working"
#define PRNTERR "Your printer is not responding"
#define AGEERR "You cannot enter an age that small"
#define NAMEERR "You must enter a full name"
```

Review Questions

The answers to the review questions are in Appendix B.



1. True or false: You can define variables with the preprocessor directives.



2. Which preprocessor directive merges another file into your program?
3. Which preprocessor directive defines literals throughout your program?
4. True or false: You can define character, string, integer, and floating-point literals with the `#define` directive.



5. Which happens first: your program is compiled or pre-processed?
6. What C++ keyword is used to replace the `#define` preprocessor directive?
7. When do you use the angled brackets in an `#include`, and when do you use double quotation marks?
8. Which are easier to change: defined literals or literals that you type throughout a program? Why?
9. Which header file should you include in almost every C++ program you write?
10. True or false: The `#define` in the following:

```
#define MESSAGE "Please press Enter to continue..."
```

changes this statement:

```
cout << "MESSAGE";
```

11. What is the output from the following program?

```
// Filename: C6EXER.C

#include <iostream.h>
#define AMT1 a+a+a
#define AMT2 AMT1 - AMT1

main()
{
    int a=1;
    cout << "Amount is " << AMT2 << "\n";
    return 0;
}
```

Even if you get this right, you will appreciate the side effects of `#define`. The `const` keyword (discussed in Chapter 4, “Variables and Literals”) before a constant variable has none of the side effects that `#define` has.

Review Exercises



1. Write a program that prints your name to the screen. Use a defined literal for the name. Do not use a character array, and don't type your actual name inside the `cout`.



2. Suppose your boss wanted you to write a program that produced an “exception report.” If the company's sales are less than \$100,000.00 or more than \$750,000.00, your boss wants your program to print the appropriate message. You learn how to produce these types of reports later in the book, but for now just write the `#define` statements that define these two floating-point literals.



3. Write the `cout` statements that print your name and birth date to the screen. Store these statements in their own file. Write a second program that includes the first file and prints your name and birth date. Be sure also to include `<iostream.h>`, because the included file contains `cout` statements.

4. Write a program that defines the ten digits, 0 through 9, as literals `ZERO` through `NINE`. Add these ten defined digits and print the result.

Summary

This chapter taught you the `#include` and `#define` preprocessor directives. Despite the fact that these directives are not executed, they temporarily change your source code by merging and defining literals into your program.

EXAMPLE

The next chapter, “Simple Input/Output,” explains input and output in more detail. There are ways to control precision when using `cin` and `cout`, as well as built-in functions that format input and output.



Simple Input/Output

You have already seen the `cout` operator. It prints values to the screen. There is much more to `cout` than you have learned. Using `cout` and the screen (the most common output device), you can print information any way you want it. Your programs also become much more powerful if you learn to receive input from the keyboard. `cin` is an operator that mirrors the `cout`. Instead of sending output values to the screen, `cin` accepts values that the user types at the keyboard.

The `cout` and `cin` operators offer the new C++ programmer input and output operators they can use with relative ease. Both of these operators have a limited scope, but they give you the ability to send output from and receive input to your programs. There are corresponding functions supplied with all C++ compilers called `printf()` and `scanf()`. These functions are still used by C++ programmers due to their widespread use in regular C programs.

This chapter introduces you to

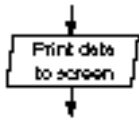
- ◆ The `cout` operator
- ◆ Control operators
- ◆ The `cin` operator

- ♦ The `printf()` output function
- ♦ The `scanf()` input function

You will be surprised at how much more advanced your programs can be after you learn these input/output operators.

The `cout` Operator

`cout` sends output to the screen.



The `cout` operator sends data to the standard output device. The standard output device is usually the screen; you can, however, redirect standard output to another device. If you are unfamiliar with device redirection at the operating system level, don't worry, you learn more about it in this book. At this point, `cout` sends all output to the screen.

The format of the `cout` is different from those of other C++ commands. The format for `cout` is

```
cout << data [ << data ];
```

The `data` placeholder can be variables, literals, expressions, or a combination of all three.

Printing Strings

To print a string constant, simply type the string constant after the `cout` operator. For example, to print the string, `The rain in Spain`, you would simply type this:



Print the sentence "The rain in Spain" to the screen.

```
cout << "The rain in Spain";
```

You must remember, however, that `cout` does not perform an automatic carriage return. This means the screen's cursor appears directly after the last printed character and subsequent `couts` begin thereafter.

To better understand this concept, try to predict the output from the following three `cout` operators:

```
cout << "Li ne 1";  
cout << "Li ne 2";  
cout << "Li ne 3";
```

These operators produce the following output:

```
Li ne 1Li ne 2Li ne 3
```

which is probably not what you intended. Therefore, you must include the newline character, `\n`, whenever you want to move the cursor to the next line. The following three `cout` operators produce a three-line output:

```
cout << "Li ne 1\n";  
cout << "Li ne 2\n";  
cout << "Li ne 3\n";
```

The output from these `cout`s is

```
Li ne 1  
Li ne 2  
Li ne 3
```

The `\n` character sends the cursor to the next line no matter where you insert it. The following three `cout` operators also produce the correct three-line output:

```
cout << "Li ne 1";  
cout << "\nLi ne 2\n";  
cout << "Li ne 3";
```

The second `cout` prints a newline before it prints anything else. It then prints its string followed by another newline. The third string prints on the third line.

You also can print strings stored in character arrays by typing the array name inside the `cout`. If you were to store your name in an array defined as:

```
char my_name[ ] = "Lyndon Harri s";
```

you could print the name with the following `cout`:

```
cout << my_name;
```

The following section of code prints three string literals on three different lines:

```
cout << "Nancy Carson\n";
cout << "1213 Oak Street\n";
cout << "Fairbanks, Alaska\n";
```

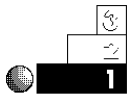
The `cout` is often used to label output. Before printing an age, amount, salary, or any other numeric data, you should print a string constant that tells the user what the number means. The following `cout` tells the user that the next number printed is an age. Without this `cout`, the user would not know what the number represented.

```
cout << "Here is the age that was found in our files:";
```

You can print a blank line by printing two newline characters, `\n`, next to each other after your string, as in:

```
cout << "Prepare the invoices...\n\n";
```

Examples



1. The following program stores a few values in three variables, then prints the results:

```
// Filename: C7PRNT1.CPP
// Prints values in variables.

#include <iostream.h>

main()
{
    char first = 'E';      // Store some character, integer,
    char middle = 'W';    // and floating-point variable.
    char last = 'C';
    int age = 32;
    int dependents = 2;
    float salary = 25000.00;
    float bonus = 575.25;

    // Prints the results.
    cout << first << middle << last;
```

```

    cout << age << dependents;
    cout << salary << bonus;
    return 0;
}

```



2. The last program does not help the user. The output is not labeled, and it prints on a single line. Here is the same program with a few messages included and some newline characters placed where needed:

```

// File name: C7PRNT2.CPP
// Prints values in variables with appropriate labels.

#include <iostream.h>

main()
{
    char first = 'E';    // Store some character, integer,
    char middle = 'W';  // and floating-point variable.
    char last = 'C';
    int age = 32;
    int dependents = 2;
    float salary = 25000.00;
    float bonus = 575.25;

    // Prints the results.
    cout << "Here are the initials:\n";
    cout << first << middle << last << "\n";
    cout << "The age and number of dependents are\n";
    cout << age << "    " << dependents << "\n\n";
    cout << "The salary and bonus are\n";
    cout << salary << "    " << bonus;
    return 0;
}

```

The output from this program appears below:

```

Here are the initials:
EWC
The age and number of dependents are
32    2

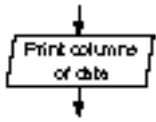
```

The salary and bonus are
25000 575.25

The first floating-point values do not print with zeros, but the number is correct. The next section shows you how to set the number of leading and trailing zeros.



3. If you have to print a table of numbers, you can use the `\t` tab character to do so. Place the tab character between each of the printed numbers. The following program prints a list of team names and number of hits for the first three weeks of the season:



```
// Filename: C7TEAM.CPP
// Prints a table of team names and hits for three weeks.
```

```
#include <iostream.h>

main()
{
    cout << "Parrots\tRams\tKings\tTitans\tChargers\n";
    cout << "3\t5\t3\t1\t0\n";
    cout << "2\t5\t1\t0\t1\n";
    cout << "2\t6\t4\t3\t0\n";
    return 0;
}
```

This program produces the table shown below. You can see that even though the names are different widths, the numbers print correctly beneath them. The `\t` character forces the next name or value to the next tab position (every eight characters).

Parrots	Rams	Kings	Titans	Chargers
3	5	3	1	0
2	5	1	0	1
2	6	4	3	0

Control Operators

You have already seen the need for additional program-output control. All floating-point numbers print with too many decimal places for most applications. What if you want to print only dollars and cents (two decimal places), or print an average with a single decimal place?

You can modify the way numbers print.

You can specify how many print positions to use in printing a number. For example, the following `cout` prints the number 456, using three positions (the length of the data):

```
cout << 456;
```

If the 456 were stored in an integer variable, it would still use three positions to print because the number of digits printed is three. However, you can specify how many positions to print. The following `cout` prints the number 456 in five positions (with two leading spaces):

```
cout << setw(5) << setfill(' ') << 456;
```

You typically use the `setw` manipulator when you want to print data in uniform columns. Be sure to include the `iomanip.h` header file in any programs that use manipulators because `iomanip.h` describes how the `setw` works to the compiler.

The following program shows you the importance of the width number. Each `cout` output is described in the comment to its left.

```
// Filename: C7MOD1.CPP
// Illustrates various integer width cout modifiers.

#include <iostream.h>
#include <iomanip.h>

main()
{
    // The output appears below.
    cout << 456 << 456 << 456 << "\n"; // Prints 456456456
    cout << setw(5) << 456 << setw(5) << 456 << setw(5) <<
        456 << "\n"; // Prints 456 456 456
    cout << setw(7) << 456 << setw(7) << 456 << setw(7) <<
        456 << "\n"; // Prints 456 456 456
    return 0;
}
```

When you use a `setw` manipulator inside a conversion character, C++ right-justifies the number by the width you specify. When you specify an eight-digit width, C++ prints a value inside those eight digits, padding the number with leading blanks if the number does not fill the whole width.



NOTE: If you do not specify a width large enough to hold the number, C++ ignores your width request and prints the number in its entirety.

You can control the width of strings in the same manner with the `setw` manipulator. If you don't specify enough width to output the full string, C++ ignores the width. The mailing list application in the back of this book uses this technique to print names on mailing labels.



NOTE: `setw()` becomes more important when you print floating-point numbers.

`setprecision(2)` prints a floating-point number with two decimal places. If C++ has to round the fractional part, it does so. The following `cout`:

```
cout << setw(6) << setprecision(2) << 134.568767;
```

produces the following output:

```
134.57
```

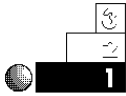
Without the `setw` or `setprecision` manipulators, C++ would have printed:

```
134.568767
```



TIP: When printing floating-point numbers, C++ always prints the entire portion to the left of the decimal (to maintain as much accuracy as possible) no matter how many positions you specify. Therefore, many C++ programmers ignore the `setw` manipulator for floating-point numbers and only specify the precision, as in `setprecision(2)`.

Examples



1. If you want to control the width of your data, use a `setw` manipulator. The following program is a revision of the `C7TEAM.CPP` shown earlier. Instead of using the tab character, `\t`, which is limited to eight spaces, this program uses the width specifier to set the tabs. It ensures that each column is 10 characters wide.

```
// Filename: C7TEAMMD.CPP
// Prints a table of team names and hits for three weeks
// using width-modifying conversion characters.

#include <iostream.h>
#include <iomanip.h>

main()
{
    cout << setw(10) << "Parrots" << setw(10) <<
        "Rams" << setw(10) << "Kings" << setw(10) <<
        "Titans" << setw(10) << "Chargers" << "\n";
    cout << setw(10) << 3 << setw(10) << 5 <<
        setw(10) << 2 << setw(10) << 1 <<
        setw(10) << 0 << "\n";
    cout << setw(10) << 2 << setw(10) << 5 <<
        setw(10) << 1 << setw(10) << 0 <<
        setw(10) << 1 << "\n";
    cout << setw(10) << 2 << setw(10) << 6 <<
        setw(10) << 4 << setw(10) << 3 <<
        setw(10) << 0 << "\n";
    return 0;
}
```



2. The following program is a payroll program. The output is in “dollars and cents” because the dollar amounts print properly to two decimal places.

```
// Filename: C7PAY1.CPP
// Computes and prints payroll data properly in dollars
// and cents.
```

Chapter 7 ♦ Simple Input/Output



```
#include <iostream.h>
#include <iomanip.h>

main()
{
    char emp_name[ ] = "Larry Payton";
    char pay_date[ ] = "03/09/92";
    int hours_worked = 43;
    float rate = 7.75;           // Pay per hour
    float tax_rate = .32;       // Tax percentage rate
    float gross_pay, taxes, net_pay;

    // Computes the pay amount.
    gross_pay = hours_worked * rate;
    taxes = tax_rate * gross_pay;
    net_pay = gross_pay - taxes;

    // Prints the results.
    cout << "As of: " << pay_date << "\n";
    cout << emp_name << " worked " << hours_worked <<
        " hours\n";
    cout << "and got paid " << setw(2) << setprecision(2)
        << gross_pay << "\n";
    cout << "After taxes of: " << setw(6) << setprecision(2)
        << taxes << "\n";
    cout << "his take-home pay was $" << setw(8) <<
        setprecision(2) << net_pay << "\n";
    return 0;
}
```

The output from this program follows. Remember that the floating-point variables still hold the full precision (to six decimal places), as they did in the previous program. The modifying `setw` manipulators only affect how the variables are output, not what is stored in them.

```
As of: 03/09/92
Larry Payton worked 43 hours
and got paid 333.25
After taxes of: 106.64
his take-home pay was $226.61
```



3. Most C++ programmers do not use the `setw` manipulator when printing dollars and cents. Here is the payroll program again that uses the shortcut floating-point width method. Notice the previous three `cout` statements include no `setw` manipulator. C++ automatically prints the full number to the left of the decimal and prints only two places to the right.

```
// Filename: C7PAY2.CPP
// Computes and prints payroll data properly
// using the shortcut modifier.

#include <iostream.h>
#include <iomanip.h>

main()
{
    char emp_name[ ] = "Larry Payton";
    char pay_date[ ] = "03/09/92";
    int hours_worked = 43;
    float rate = 7.75;           // Pay per hour
    float tax_rate = .32;       // Tax percentage rate
    float gross_pay, taxes, net_pay;

    // Computes the pay amount.
    gross_pay = hours_worked * rate;
    taxes = tax_rate * gross_pay;
    net_pay = gross_pay - taxes;

    // Prints the results.
    cout << "As of: " << pay_date << "\n";
    cout << emp_name << " worked " << hours_worked <<
        " hours\n";
    cout << "and got paid " << setprecision(2) << gross_pay
        << "\n";
    cout << "After taxes of: " << setprecision(2) << taxes
        << "\n";
    cout << "his take-home pay was " << setprecision(2) <<
        net_pay << "\n";
    return 0;
}
```

This program's output is the same as the previous program's.

The `cin` Operator

You now understand how C++ represents data and variables, and you know how to print the data. There is one additional part of programming you have not seen: inputting data to your programs.

Until this point, you have not inputted data into a program. All data you worked with was assigned to variables in the program. However, this is not always the best way to transfer data to your programs; you rarely know what your data is when you write your programs. The data is known only when you run the programs (or another user runs them).

The `cin` operator is one way to input from the keyboard. When your programs reach the line with a `cin`, the user can enter values directly into variables. Your program can then process those variables and produce output. Figure 7.1 illustrates the difference between `cout` and `cin`.

The `cin` operator stores keyboard input in variables.

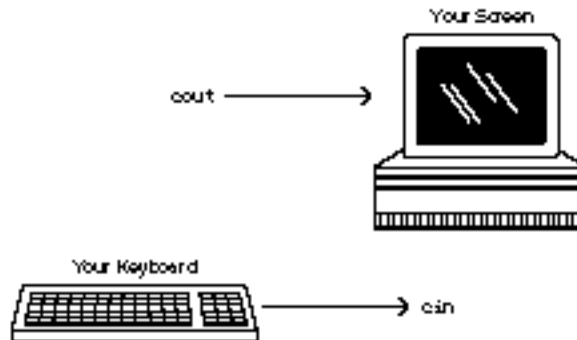


Figure 7.1. The actions of `cout` and `cin`.

The `cin` Function Fills Variables with Values

There is a major difference between `cin` and the assignment statements (such as `i = 17;`). Both fill variables with values. However, the assignment statement assigned specific values to variables at programming time. When you run a program with assignment statements, you know from the program's listing exactly what values go into the variables because you wrote the program specifically to store those values. Every time you run the program, the results are exactly the same because the same values are *assigned* to the same variables.

You have no idea, when you write programs that use `cin`, what values will be assigned to the `cin`'s variables because their values are not known until the program runs and the user enters those values. This means you have a more flexible program that can be used by a variety of people. Every time the program is run, different results are created, depending on the values typed at each `cin` in the program.

The `cin` has its drawbacks. Therefore, in the next few chapters you will use `cin` until you learn more powerful (and flexible) input methods. The `cin` operator looks much like `cout`. It contains one or more variables that appear to the right of the operator name. The format of the `cin` is

```
cin >> value [>> values];
```

The `iostream.h` header file contains the information C++ needs to use `cin`, so include it when using `cin`.

NOTE: The `cin` operator uses the same manipulators (`setw` and `setprecision`) as the `cout` operator.

As mentioned earlier, `cin` poses a few problems. The `cin` operator requires that your user type the input *exactly* as `cin` expects it. Because you cannot control the user's typing, this cannot be ensured. You might want the user to enter an integer value followed



by a floating-point value and your `cin` operator call might expect it too, but your user might decide to enter something else! If this happens, there is not much you can do because the resulting input is incorrect and your C++ program has no reliable method for testing user accuracy. Before every `cin`, print a prompt that explains exactly what you expect the user to type.

The `cin` operator requires that the user type correct input. This is not always possible to guarantee!

For the next few chapters, you can assume that the user knows to enter the proper values, but for your “real” programs, read on for better methods to receive input, starting with Chapter 21, “Device and Character Input/Output.”

Examples



1. If you wanted a program that computed a seven percent sales tax, you could use the `cin` statement to figure the sales, compute the tax, and print the results as the following program shows:

```

// Filename: C7SLTX1.CPP
// Prompt for a sales amount and print the sales tax.

#include <iostream.h>
#include <iomanip.h>

main()
{
    float total_sale;    // User's sale amount goes here.
    float stax;

    // Display a message for the user.
    cout << "What is the total amount of the sale? ";

    // Receive the sales amount from user.
    cin >> total_sale;

    // Calculate sales tax.
    stax = total_sale * .07;
  
```



```

cout << "The sales tax for " << setprecision(2) <<
    total_sale << " is " << setprecision(2) << stax;
return 0;
}

```

Because the first `cout` does not contain a newline character, `\n`, the user's response to the prompt appears to the right of the question mark.



- When inputting keyboard strings into character arrays with `cin`, you are limited to receiving one word at a time. The `cin` does not enable you to type more than one word in a single character array at a time. The following program asks the user for his or her first and last name. The program has to store those two names in two different character arrays because `cin` cannot input both names at once. The program then prints the names in reverse order.

```

// Filename: C7PHON1.CPP
// Program that requests the user's name and prints it
// to the screen as it would appear in a phone book.

#include <iostream.h>
#include <iomanip.h>

main()
{
    char first[20], last[20];

    cout << "What is your first name? ";
    cin >> first;
    cout << "What is your last name? ";
    cin >> last;
    cout << "\n\n"; // Prints two blank lines.
    cout << "In a phone book, your name would look like this:\n";
    cout << last << ", " << first;
    return 0;
}

```



3. Suppose you want to write a program that does simple addition for your seven-year-old daughter. The following program prompts her for two numbers. The program then waits for her to type an answer. When she gives her answer, the program displays the correct result so she can see how well she did.

```
// Filename: C7MATH.CPP
// Program to help children with simple addition.
// Prompt child for two values after printing
// a title message.
#include <iostream.h>
#include <iomanip.h>

main()
{
    int num1, num2, ans;
    int her_ans;

    cout << "*** Math Practice ***\n\n\n";
    cout << "What is the first number? ";
    cin >> num1;
    cout << "What is the second number? ";
    cin >> num2;

    // Compute answer and give her a chance to wait for it.
    ans = num1 + num2;

    cout << "\nWhat do you think is the answer? ";
    cin >> her_ans;    // Nothing is done with this.

    // Prints answer after a blank line.
    cout << "\n" << num1 << " plus " << num2 << " is "
        << ans << "\n\nHope you got it right!";
    return 0;
}
```

printf() and scanf()

Before C++, C programmers had to rely on function calls to perform input and output. Two of those functions, `printf()` and `scanf()`, are still used frequently in C++ programs, although `cout` and `cin` have advantages over them. `printf()` (like `cout`) prints values to the screen and `scanf()` (like `cin`) inputs values from the keyboard. `printf()` requires a controlling format string that describes the data you want to print. Likewise, `scanf()` requires a controlling format string that describes the data the program wants to receive from the keyboard.



NOTE: `cout` is the C++ replacement to `printf()` and `cin` is the C++ replacement to `scanf()`.

Because you are concentrating on C++, this chapter only briefly covers `printf()` and `scanf()`. Throughout this book, a handful of programs use these functions to keep you familiar with their format. `printf()` and `scanf()` are not obsolete in C++, but their use will diminish dramatically when programmers move away from C and to C++. `cout` and `cin` do not require controlling strings that describe their data; `cout` and `cin` are intelligent enough to know how to treat data. Both `printf()` and `scanf()` are limited—especially `scanf()`—but they do enable your programs to send output and to receive input.

The printf() Function

The `printf()` function sends output to the screen.

`printf()` sends data to the standard output device, which is generally the screen. The format of `printf()` is different from those of regular C++ commands. The values that go inside the parentheses vary, depending on the data you are printing. However, as a general rule, the following `printf()` format holds true:

```
printf(control_string [, one or more values]);
```

Notice `printf()` always requires a `control_string`. This is a string, or a character array containing a string, that determines how the rest of the values (if any are listed) print. These values can be variables, literals, expressions, or a combination of all three.



TIP: Despite its name, `printf()` sends output to the screen and not to the printer.

The easiest data to print with `printf()` are strings. To print a string constant, you simply type that string constant inside the `printf()` function. For example, to print the string `The rain in Spain`, you would simply type the following:



Print the phrase “The rain in Spain” to the screen.

```
printf("The rain in Spain");
```

`printf()`, like `cout`, does *not* perform an automatic carriage return. Subsequent `printf()`s begin next to that last printed character. If you want a carriage return, you must supply a newline character, as so:

```
printf("The rain in Spain\n");
```

You can print strings stored in character arrays also by typing the array name inside the `printf()`. For example, if you were to store your name in an array defined as:

```
char my_name[] = "Lyndon Harris";
```

you could print the name with this `printf()`:

```
printf(my_name);
```

You must include the `stdio.h` header file when using `printf()` and `scanf()` because `stdio.h` determines how the input and output functions work in the compiler. The following program assigns a message in a character array, then prints that message.

```
// Filename: C7PS2.CPP
// Prints a string stored in a character array.
#include <stdio.h>
main()
{
    char message[] = "Please turn on your printer";
    printf(message);
    return 0;
}
```

Conversion Characters

Inside most `printf()` control strings are *conversion characters*. These special characters tell `printf()` exactly how the data (following the characters) are to be interpreted. Table 7.1 shows a list of common conversion characters. Because any type of data can go inside the `printf()`'s parentheses, these conversion characters are required any time you print more than a single string constant. If you don't want to print a string, the string constant must contain at least one of the conversion characters.

Table 7.1. Common `printf()` conversion characters.

<i>Conversion Character</i>	<i>Output</i>
<code>%s</code>	String of characters (until null zero is reached)
<code>%c</code>	Character
<code>%d</code>	Decimal integer
<code>%f</code>	Floating-point numbers
<code>%u</code>	Unsigned integer
<code>%x</code>	Hexadecimal integer
<code>%%</code>	Prints a percent sign (%)

Note: You can insert an `l` (lowercase `l`) or `L` before the integer and floating-point conversion characters (such as `%ld` and `%Lf`) to indicate that a long integer or long double floating-point is to be printed.



NOTE: Characters other than those shown in the table print exactly as they appear in the control string.

When you want to print a numeric constant or variable, you must include the proper conversion character inside the `printf()` control string. If `i`, `j`, and `k` are integer variables, you cannot print them with the `printf()` that follows.

```
printf(i, j, k);
```

Because `printf()` is a function and not a command, this `printf()` function has no way of knowing what type the variables are. The results are unpredictable, and you might see garbage on your screen—if anything appears at all.

When you print numbers, you must first print a control string that includes the format of those numbers. The following `printf()` prints a string. In the output from this line, a string appears with an integer (`%d`) and a floating-point number (`%f`) printed inside that string.

```
printf("I am Betty, I am %d years old, and I make %f\n",
      35, 34050.25);
```

This produces the following output:

```
I am Betty, I am 35 years old, and I make 34050.25
```

Figure 7.2 shows how C interprets the control string and the variables that follow. Be sure you understand this example before moving on. It is the foundation of the `printf()` function.

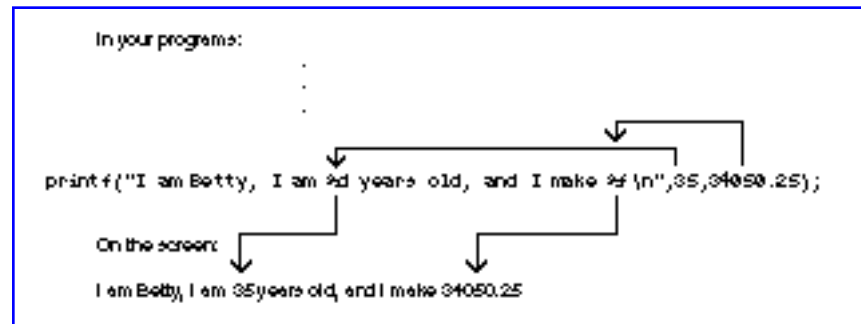
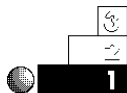


Figure 7.2. Control string in action.

You also can print integer and floating-point variables in the same manner.

Examples

1. The following program stores a few values in three variables, then prints the results.



```
// Filename: C7PRNTF.CPP
// Prints values in variables with appropriate labels.
#include <stdio.h>

main()
{
    char first='E';           // Store some character, integer,
    char middle='W';         // and floating-point variable.
    char last='C';
    int age=32;
    int dependents=2;
    float salary=25000.00;
    float bonus=575.25;

    /* Prints the results. */
    printf("Here are the initials\n");
    printf("%c%c%c\n\n", first, middle, last);
    printf("The age and number of dependents are\n");
    printf("%d %d\n\n", age, dependents);
    printf("The salary and bonus are\n");
    printf("%f %f", salary, bonus);
    return 0;
}
```

The output from this program is

```
Here are the initials
EWC
```

```
The age and number of dependents are
32  2
```

```
The salary and bonus are
25000.000000 575.250000
```



2. The floating-point values print with too many zeros, of course, but the numbers are correct. You can limit the number of leading and trailing zeros that is printed by adding a width specifier in the control string. For instance, the following `printf()` prints the salary and bonus with two decimal places:

```
printf("%.2f %.2f", salary, bonus);
```

Make sure your printed values match the control string supplied with them. The `printf()` function cannot fix problems resulting from mismatched values and control strings. Don't try to print floating-point values with character-string control codes. If you list five integer variables in a `printf()`, be sure to include five `%d` conversion characters in the `printf()` as well.

Printing ASCII Values

There is one exception to the rule of printing with matching conversion characters. If you want to print the ASCII value of a character, you can print that character (whether it is a constant or a variable) with the integer `%d` conversion character. Instead of printing the character, `printf()` prints the matching ASCII number for that character.

Conversely, if you print an integer with a `%c` conversion character, you see the character that matches that integer's value from the ASCII table.

The following `printf()`s illustrate this fact:

```
printf("%c", 65); // Prints the letter A.
printf("%d", 'A'); // Prints the number 65.
```

The `scanf()` Function

The `scanf()` function stores keyboard input to variables.

The `scanf()` function reads input from the keyboard. When your programs reach the line with a `scanf()`, the user can enter values directly into variables. Your program can then process the variables and produce output.

The `scanf()` function looks much like `printf()`. It contains a control string and one or more variables to the right of the control string. The control string informs C++ exactly what the incoming keyboard values look like, and what their types are. The format of `scanf()` is

```
scanf(control_string, one or more values);
```


The `scanf()` `control_string` uses almost the same conversion characters as the `printf()` `control_string`, with two slight differences. You should never include the newline character, `\n`, in a `scanf()` control string. The `scanf()` function “knows” when the input is finished when the user presses Enter. If you supply an additional newline code, `scanf()` might not terminate properly. Also, always put a beginning space inside every `scanf()` control string. This does not affect the user’s input, but `scanf()` sometimes requires it to work properly. Later examples in this chapter clarify this fact.

The `scanf()` function requires that your user type accurately. This is not always possible to guarantee!

As mentioned earlier, `scanf()` poses a few problems. The `scanf()` function requires that your user type the input exactly the way `control_string` specifies. Because you cannot control your user’s typing, this cannot always be ensured. For example, you might want the user to enter an integer value followed by a floating-point value (your `scanf()` control string might expect it too), but your user might decide to enter something else! If this happens, there is not much you can do. The resulting input is incorrect, but your C program has no reliable method for testing user accuracy before your program is run.



CAUTION: The user’s keyboard input values *must* match, in number and type, the control string contained in each `scanf()`.

Another problem with `scanf()` is not as easy for beginners to understand as the last. The `scanf()` function requires that you use pointer variables, not regular variables, in its parentheses. Although this sounds complicated, it doesn’t have to be. You should have no problem with `scanf()`’s pointer requirements if you remember these two simple rules:

1. Always put an ampersand (&) before variable names inside a `scanf()`.
2. Never put an ampersand (&) before an array name inside a `scanf()`.

Despite these strange `scanf()` rules, you can learn this function quickly by looking at a few examples.

Examples



1. If you want a program that computes a seven percent sales tax, you could use the `scanf()` statement to receive the sales, compute the tax, and print the results as the following program shows.

```
// Filename: C7SLTXS.CPP
// Compute a sales amount and print the sales tax.
#include <stdio.h>
main()
{
    float total_sale; // User's sale amount goes here.
    float stax;

    // Display a message for the user.
    printf("What is the total amount of the sale? ");

    // Compute the sales amount from user.
    scanf(" %f", &total_sale); // Don't forget the beginning
                                // space and an &.

    stax = total_sale * .07; // Calculate the sales tax.

    printf("The sales tax for %.2f is %.2f", total_sale, stax);
    return 0;
}
```

If you run this program, the program waits for you to enter a value for the total sale. Remember to use the ampersand in front of the `total_sale` variable when you enter it in the `scanf()` function. After pressing the Enter key, the program calculates the sales tax and prints the results.

If you entered 10.00 as the sale amount, you would receive the following output :

```
The sales tax for 10.00 is 0.70
```



2. Use the string `%s` conversion character to input keyboard strings into character arrays with `scanf()`. As with `cin`, you are limited to inputting one word at a time, because you

cannot type more than one word into a single character array with `scanf()`. The following program is similar to `C7PHON1.CPP` except the `scanf()` function, rather than `cin`, is used. It must store two names in two different character arrays, because `scanf()` cannot input both names at once. The program then prints the names in reverse order.

```
// Filename: C7PHON2.CPP
// Program that requests the user's name and prints it
// to the screen as it would appear in a phone book.
#include <stdio.h>
main()
{
    char first[20], last[20];
    printf("What is your first name? ");
    scanf(" %s", first);
    printf("What is your last name? ");
    scanf(" %s", last);
    printf("\n\n");    // Prints two blank lines.
    printf("In a phone book, your name would look like"
           " this: \n");
    printf("%s, %s", last, first);
    return 0;
}
```

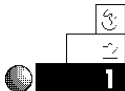
3. How many values are entered with the following `scanf()`, and what are their types?

```
scanf(" %d %d %f %s", &i, &j, &k, l);
```

Review Questions

The answers to the Review Questions are in Appendix B.

1. What is the difference between `cout` and `cin`?
2. Why is a prompt message important before using `cin` for input?



3. How many values do you enter with the following `cin`?

```
cin >> i >> j >> k >> l;
```



4. Because they both assign values to variables, is there any difference between assigning values to variables and using `cin` to give them values?

5. True or false: The `%s` conversion character is usually not required in `printf()` control strings.

6. Which types of variables do not require the ampersand (&) character in `scanf()` functions?



7. What is the output produced by the following `cout`?

```
cout << "The backslash \"\\\" character is special";
```

8. What is the result of the following `cout`?

```
cout << setw(8) << setprecision(3) << 123.456789;
```

Review Exercises



1. Write a program that prompts the user for his or her name and weight. Store these values in separate variables and print them on-screen.

2. Assume you are a college professor and have to average grades for 10 students. Write a program that prompts you for 10 different grades, then displays an average of them.



3. Modify the program in Exercise 2 to ask for each student's name as well as her grade. Print the grade list to the screen, with each student's name and grade in two columns. Make sure the columns align by using a `setw` manipulator on the grade. At the bottom, print the average of the grades. (*Hint:* Store the 10 names and 10 grades in different variables with different names.) This program is easy, but takes thirty or so lines, plus appropriate comments and prompts. Later, you learn ways to streamline this program.



4. This exercise tests your understanding of the backslash conversion character: Write a program that uses `cout` operators to produce the following picture on-screen:

```

          +
        /*\
         |||
         |||
         |||
         |||
        /|||\
       / * \
      ***
      *
_____/_+|||+_|_____/_=====\\_____
| + + |
| || |
/ |=====\
 / \ ||
 ^ \ _ *
 **
 *
```

Summary

You now can print almost anything to the screen. By studying the manipulators and how they behave, you can control your output more thoroughly than ever before. Also, because you can receive keyboard values, your programs are much more powerful. No longer do you have to know your data values when you write the program. You can ask the user to enter values into variables with `cin`.

You have the tools to begin writing programs that fit the data processing model of INPUT->PROCESS->OUTPUT. This chapter concludes the preliminary discussion of the C++ language. This part of the book attempted to give you an overview of the language and to teach you enough of the language elements so you can begin writing helpful programs.

Chapter 8, “Using C++ Math Operators and Precedence,” begins a new type of discussion. You learn how C++’s math and relational operators work on data, and the importance of the precedence table of operators.

