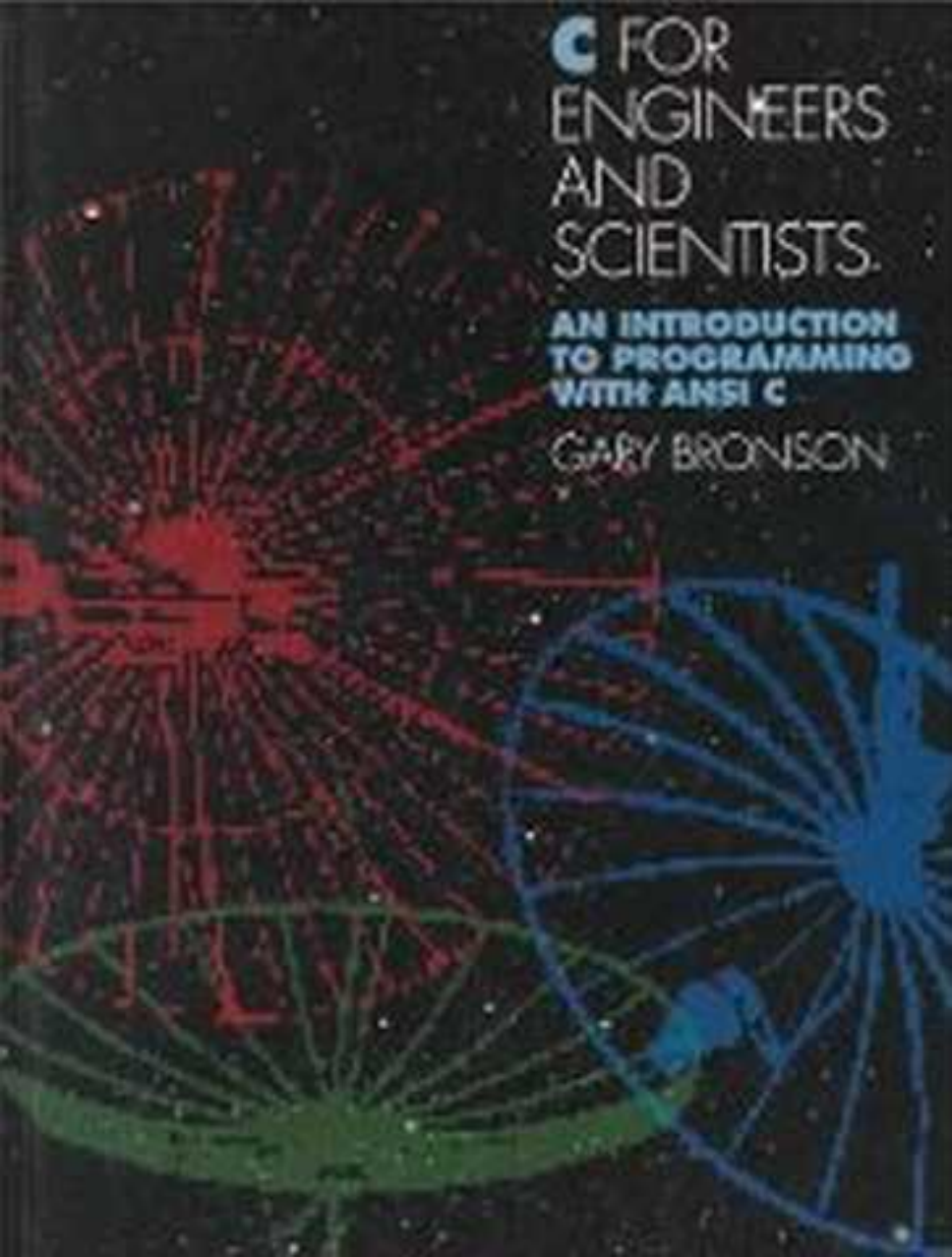
C FOR ENGINEERS AND SCIENTISTS

AN INTRODUCTION TO PROGRAMMING WITH ANSI C

GARY BRONSON



C for Engineers and Scientists

An Introduction to Programming with ANSI C

Gary J. Bronson
Fairleigh Dickinson University

Howard Silver
consulting editor

West Publishing Company
Minneapolis/Saint Paul New York/Los Angeles San Francisco

FREE COPY

PRODUCTION CREDITS

Text design: *David Farr, Imagesmythe, Inc.*
Copyeditor: *Sheryl Rose*
Artwork: *Alexander Teshin Associates*
Composition: *The Clarinda Company*

West's Commitment to the Environment

In 1906, West Publishing Company began recycling materials left over from the production of books. This began a tradition of efficient and responsible use of resources. Today, up to 95 percent of our legal books and 70 percent of our college and school texts are printed on recycled, acid-free stock. West also recycles nearly 22 million pounds of scrap paper annually—the equivalent of 181,717 trees. Since the 1960s, West has devised ways to capture and recycle waste inks, solvents, oils, and vapors created in the printing process. We also recycle plastics of all kinds, wood, glass, corrugated cardboard, and batteries, and have eliminated the use of styrofoam book packaging. We at West are proud of the longevity and the scope of our commitment to our environment.

Production, Prepress, Printing and Binding by West Publishing Company.

Copyright ©1993 By WEST PUBLISHING COMPANY
610 Opperman Drive
P.O. Box 64526
St. Paul, MN 55164-0526

All rights reserved

Printed in the United States of America

00 99 98 97 96 95 94 93 8 7 6 5 4 3 2 1 0

Library of Congress Cataloging-in-Publication Data

Bronson, Gary J.
C for engineers and scientists: an introduction to programming / Gary Bronson.
p. cm.
Includes index.
ISBN 0-314-00816-0 (soft)
1. C (Computer program language) I. Title.
QA76.73.C15B738 1993
005.13'3—dc20 92-21688
CIP

dedicated to

Rochelle,
Jeremy,
David, and
Matthew

FREE COPY

Preface

The C programming language has become the preeminent applications language of the 1990s in almost every area except the engineering sciences. There is an irony in this because C was developed in 1972 at ATT Bell Laboratories, a world-renowned engineering and scientific facility. Slowly, however, C has taken a foothold in the engineering and science world and is becoming acknowledged within the engineering and scientific community as an accepted programming language on a par with FORTRAN. This has happened for a variety of reasons and, interestingly, mirrors the acceptance of C as the standard programming language used within Bell Laboratories.

Since the C programming language was initially designed in conjunction with the development of the UNIX operating system, it became known as an operating systems language, even within Bell Laboratories. In the late 1970s, however, the engineers and scientists at ATT started to notice that C had characteristics that also made it extremely useful for engineering and scientific purposes. These characteristics are:

Efficiency. Compiled C code tends to be compact and extremely fast in execution. As C also provides the ability to do bit manipulations and directly designate and access memory storage, which are traditionally only provided by assembly languages, it provides the programmer with a degree of fine-tuning.

Structure. As C was developed after the major emphasis and research on structured programming and software engineering done in the 1960s, it incorporated the control features developed by this research. Thus, C supports top-down program design and a modular program structure, which makes programs easier to read, debug, and modify.

Portability. Portability determines the degree to which a program written on one computer system can be moved to any other system. Generally, C source code can be moved across computer systems with little or no modification. For example, C source code written on a UNIX based machine can usually be compiled on an IBM PC or mainframe with almost no modifications required in the code. With the availability of inexpensive compilers that run on personal computers, this portability has meant that programs developed at home can be used on PCs, minis, and mainframes at work.

Versatility. Depending on the user's orientation, C programs can be written for real-time process control, for engineering analysis work, for statistical applications, to develop database applications, to construct operating systems, or to write FORTRAN compilers and BASIC interpreters.

For these reasons, as more and more people have tried C, they have found it to be a professional language having an unusual range of capabilities that is enjoyable to work with. Thus, C has joined FORTRAN and COBOL as a major programing language for industry.

In recognition of the importance and widespread scientific, engineering, and commercial acceptance of C, the American National Standards Institute (ANSI) began establishing a standard for the language, similar to standards previously established for FORTRAN and COBOL. As noted in the foreword to the 1989 draft of the ANSI standard, "The need for a single, clearly-defined standard had arisen in the C community due to a rapidly expanding use of the C programming language and the variety of differing translator implementations that had been and were being developed." The standard was officially ratified as American National Standard X3.159-1989 on December 14, 1989 and subsequently published in the Spring of 1990.

Because of C's growing importance in the applications and academic areas, the predecessor of this text, *A First Book of C*, was written. The primary purpose of this earlier edition was to make C more accessible as an applications programming language than was possible with texts that focused on C as an advanced operating systems language. A second edition of *A First Book of C* included expanded coverage of software engineering concepts. This permitted the book to be used as an introduction to programming in addition to being an introduction to C. This text retains the style and teaching orientation of its predecessor but is specifically oriented toward engineering and scientific applications.

Distinctive Features of This Book

Writing Style. I firmly believe that introductory texts do not teach students — professors teach students. An introductory textbook, if it is to be useful, must be the primary "supporting actor" to the "leading role" of the professor. Once the professor sets the stage, however, the textbook must encourage, nurture, and assist the student in acquiring and "owning" the material presented in class. To do this the text must be written in a manner that makes sense to the

student. One of the distinctive features of this book is that it has been written for the student. As one of the reviewers has said, "This book addresses the student and not the professional." Thus, first and foremost, I feel the writing style used to convey the concepts presented is the most important aspect of the text.

Modularity. C, by its nature, is a modular language. Thus, the connection between C functions and modules is made early in the text, in Section 1.2, and continues throughout the book. To stress the modular nature of C, the first complete `main()` function illustrates calling four other functions. The first program that can be compiled is then presented, which calls the `printf()` function.

The idea of argument passing into modules is also made early, in Section 1.3, with the use of the `printf()` function. In this manner, students are introduced to functions and argument passing as a natural technique of programming.

Software Engineering. Rather than simply introduce students to C, this text introduces students to the fundamentals of software engineering. This introduction begins with Section 1.1, which introduces algorithms and the various ways that an algorithm can be described. An example illustrating three algorithms for summing the numbers from 1 to 100 (Figure 1–4) is used to make the discussion of algorithms more tangible to students.

The increased emphasis on software engineering is supported by a section (Section 1.5) on top-down program development. Here the importance of understanding the problem and selecting an appropriate algorithm is highlighted and the relationship between analysis, design, coding, and testing introduced. Problem solving within this context is stressed throughout the text.

Applications. Engineering and scientific examples are used throughout the text to illustrate the concepts presented. In addition, the majority of the chapters have a section consisting of two specific applications relating to the material presented in the chapter. Many of the applications are of the "tried and true" variety and are not unique to this book. However, some interesting new applications have been added, such as the study of acid rain, the calculation of pollen counts, the operation of telephone switching networks, and the construction of a user-written random number generator that are not typically found in introductory texts. Additionally, Chapter 9 is completely devoted to numerical applications and is a mini-introduction to numerical techniques in and of itself.

Emphasis on ANSI C. Although ANSI C is emphasized throughout the text, pre-ANSI C constructs are also shown for those of you who will be using a non-ANSI compiler. Generally, the major difference of note is in Chapter 6, where user-written functions are introduced. For these functions ANSI C requires a single function header line that includes argument declarations, while pre-ANSI compilers require argument declarations on a separate line. Also, the declaration of functions in ANSI C, called function prototypes, includes the data types of all arguments, while pre-ANSI C omits the argument declarations.

Introduction to Pointers. One of the unique features of this text is its method of introducing pointers. This is done by first using the `printf()` function to

display a variable's address so that the student can "see" what an address is. This approach, which was used in a *A First Book of C*, always seemed a more logical and intuitive method of understanding pointers than the alternative indirection description with no display.

Since the publication of *A First Book of C*, the use of the `printf()` function to display addresses has become a standard way of introducing pointers. Although this approach, therefore, is no longer a unique feature of the book, I am very proud of its presentation, and continue to use it in this text.

Program Testing. Every single C program in this text has been successfully compiled and run under Borland's Turbo C Compiler. The programs have been written using features fully supported under ANSI C. *A source diskette of all programs is available to adopters.*

Pedagogical Features

To facilitate my goal of making C accessible as a first level course, I have continued to use the following pedagogical features:

End of Section Exercises. Almost every section in the book contains numerous and diverse skill builder and programming exercises. Additionally, solutions to selected exercises are provided in an appendix.

Pseudocode and Flowchart Descriptions. As in *A First Book of C*, pseudocode is stressed throughout the text. Although flowcharts were used in *A First Book of C*, no explicit definition or introduction to flowchart symbols was presented. In this edition we have added additional material on flowchart symbols and the use of flowcharts in visually presenting flow-of-control constructs.

Common Programming Errors and Chapter Review. Each chapter ends with a section on common programming errors and a review of the main topics covered.

Appendices and Supplements

An expanded set of appendices has been provided in *C for Engineers and Scientists.* In addition to the three appendices taken from *A First Book of C* on Operator Precedence, ASCII codes, and I/O-Standard Error Redirection, the new appendices contain material on Program Life Cycle; using the DOS, UNIX, VAX, and PRIME operating systems; using Borland's Turbo C Compiler; and using Microsoft's C Compiler.

Additionally, a printed solutions manual is available containing solutions (with comments) to programming exercises not included in Appendix H of the text. A source diskette of solutions to all programs in the book is available to adoptors of the text.

Acknowledgments

C for Engineers and Scientists is a direct result of the success (and the limitations) of its predecessor, *A First Book of C*. In this regard, my most heartfelt acknowledgment and appreciation goes to the instructors and students who found the earlier edition to be of service to them in their respective quests to teach and learn C.

Once an edition devoted to engineering and scientific applications was planned, its completion depended on many people other than myself. For this I especially want to thank the staff of West Publishing Company for their many contributions. These included the continuous faith and encouragement of my editor, Richard Mixter, and developmental editor, Keith Dodson, and the many suggestions and enthusiastic work of the production editor, Thomas Modl, and promotion manager, Ellen Stanton.

The direct encouragement and support of my dean at Fairleigh Dickinson University, Dr. Paul Lerman, must also be acknowledged. Without his support and the support of my chairman, Dr. G. Naadimuthu, this text could not have been written. I also wish to express my gratitude to the individual reviewers listed on the next page. Each of these people provided suggestions and critical comments that have made an invaluable contribution to the quality of this text.

Finally, I wish to acknowledge the patience, understanding, and love provided by my wife and partner, Rochelle.

Gary Bronson

The author thanks these reviewers for their knowledgeable help in the completion of this book.

Farzan Abdolsalami
Trinity University

Farrokh Attarzadeh
University of Houston—University Park

Lorraine Callahan
(formerly) *Northern Arizona University*

Katy Disney
Mission College

David Eberly
University of North Carolina at Chapel Hill

Rhonda Ficek
Moorhead State University

Roy J. Fuller
University of Arkansas at Fayetteville

Gordon W. Hoagland
Ricks College

Paul I-Hai Lin
Indiana University/Purdue University at Fort Wayne

Robert D. Logcher
Massachusetts Institute of Technology

Luegina C. Mounfield
Louisiana State University

David Richards
Loyola College

Howard Silver
Fairleigh Dickinson University

Susan M. Simons
Memphis State University

Neil R. Sorensen
Weber State University

Donald Yee
Mesa Community College

Applications

Pendulum Clocks
Telephone Switching Networks
Acid Rain
Sine Approximations
Solving Quadratic Equations
Data Validation
Curve Plotting
Data Scaling
Random Network Generation
Coin Toss Simulation
Pollen Counting
Master/Transaction File Updating
Root Finding
Numerical Integration
Average and RMS Values
Battery Charging
Linked List Maintenance
Dynamic Storage Allocation
Data Encryption

Contents

CHAPTER ONE Getting Started 3

1.1 Introduction to Programming 4
- Algorithms 6
- From Algorithms to Programs 8
- Program Translation 9

1.2 Introduction to Modularity 12
- Functions 13
- The `main()` Function 15

1.3 The `printf()` Function 18

1.4 Programming Style 22
- Comments 23

1.5 Common Programming Errors 25

1.6 Chapter Summary 26

1.7 Enrichment Study: Computer Hardware and Storage 27

CHAPTER TWO Data Types and Operations 32

2.1 Data Constants and Arithmetic Operations 33
- Integer Constants 33
- Floating Point and Double Precision Constants 33
- Character Constants 35
- Escape Sequences 36
- Arithmetic Operations 37
- Integer Division 40
- A Unary Operator (Negation) 40
- Operator Precedence and Associativity 40

2.2 Variables and Declaration Statements 45
Declaration Statements 47
Multiple Declarations 49
Specifying Storage Allocation 51
2.3 Assignment Statements 55
Assignment Variations 59
Accumulating 60
Counting 62
2.4 Formatted Output 69
Format Modifiers 72
Other Number Bases 73
2.5 Top-Down Program Development 78
Step 1: Determine the Desired Output 79
Step 2: Determine the Input Items 79
Step 3a: Determine an Algorithm 80
Step 3b: Do a Hand Calculation 80
Step 3c: Select Variable Names 80
Step 4: Write the Program 80
Step 5: Test the Output 81
Modularity and Top-Down Design 81
2.6 Applications 85
Application 1: Pendulum Clocks 85
Application 2: Telephone Switching Networks 87
2.7 Common Programming Errors 92
2.8 Chapter Summary 93
2.9 Enrichment Study: Variable Addresses and Memory Storage 94

CHAPTER THREE Completing the Basics 96

3.1 Mathematical Library Functions 97
Type Conversions 102
3.2 The `scanf()` Function 107
3.3 `scanf()` with Buffered Input 119
3.4 Symbolic Constants 122
3.5 Applications 127
Application 1: Acid Rain 128
Application 2: Sine Approximation 130
3.6 Common Programming Errors 136
3.7 Chapter Summary 137
3.8 Enrichment Study: Program Life Cycle 138

CHAPTER FOUR Selection 143

4.1 Relational Expressions 144
Logical Operators 146
A Numerical Accuracy Problem 149

4.2 The `if-else` Statement 150
Compound Statements 152
One-Way Selection 154
Caution 155
4.3 Nested `if` Statements 161
The `if-else` Chain 162
4.4 The `switch` Statement 172
4.5 Applications 177
Application 1: Solving Quadratic Equations 177
Application 2: Data Validation 181
4.6 Common Programming Errors 185
4.7 Chapter Summary 187
4.8 Enrichment Study: A Closer Look at Errors, Testing, and Debugging 189

CHAPTER FIVE Repetition 194

5.1 The `while` Statement 195
5.2 `scanf()` Within a `while` Loop 203
Sentinels 208
`break` and `continue` Statements 212
The Null Statement 213
5.3 The `for` Statement 216
5.4 `for` Loop Programming Techniques 225
Technique 1: Interactive Input Within a `for` Loop 225
Technique 2: Selection Within a `for` Loop 226
Technique 3: Evaluating Functions of One Variable 226
Technique 4: Interactive Loop Control 229
5.5 Nested Loops 231
Program Analysis 233
5.6 The `do` Statement 235
Validity Checks 236
5.7 Common Programming Errors 238
5.8 Chapter Summary 239

CHAPTER SIX Arrays 240

6.1 One-Dimensional Arrays 241
Input and Output of Array Values 245
6.2 Array Initialization 250
6.3 Two-Dimensional Arrays 253
Larger-Dimensional Arrays 257
6.4 Applications 259
Application 1: Curve Plotting 259
Application 2: Data Scaling 263
6.5 Common Programming Errors 268
6.6 Chapter Summary 268
6.7 Enrichment Study: Sorting Methods 269

CHAPTER SEVEN Writing Your Own Functions 275

7.1 Function Definitions and Declarations 276
Defining a Function 277
Declaring a Function 280
Standard Library Function 285
Input/Output Library Function 286
String Library Functions 286
Miscellaneous Routines 287
7.2 Arrays as Arguments 291
7.3 Applications 298
Application 1: Random Number Generation 298
Application 2: Coin Toss Simulation 300
7.4 Variable Scope 304
Misuse of Globals 307
7.5 Variable Storage Class 310
Local Variable Storage Classes 310
Global Variable Classes 313
7.6 Common Programming Errors 317
7.7 Chapter Summary 318
7.8 Enrichment Study: Programming Costs 319

CHAPTER EIGHT Data Files 323

8.1 Creating and Using Data Files 324
Writing to a File 327
Closing a File 328
Reading a File 329
Standard Device Files 332
Other Devices 334
8.2 Random File Access 337
8.3 Applications 340
Application 1: Pollen Counts 340
Application 2: Master/Transaction File Update 343
8.4 Text and Binary Files 348
Text File Storage 348
Binary Files 350
8.5 Common Programming Errors 352
8.6 Chapter Summary 353
8.7 Enrichment Study: Control Codes 354

CHAPTER NINE Numerical Techniques and Applications 357

9.1 Solving Simultaneous Linear Equations 358
Two Linear Equations with Two Unknowns 358
9.2 Root Finding 369
Fixed Increment Iterations 370
The Bisection (Binary Search) Method 377
Secant Method 381

9.3 Numerical Integration 388
Rectangular Approximations 388
Modified Rectangular Approximations 392
Trapezoidal Approximations 395
Simpson's Method 398
Application: Finding Average and RMS Values 402
9.4 Common Programming Errors 407
9.5 Chapter Summary 408

CHAPTER TEN Pointers 410

10.1 Addresses and Pointers 411
Storing Addresses 413
Using Addresses 414
Declaring Pointers 415
10.2 Returning Multiple Function Arguments 421
10.3 Array Names as Pointers 428
10.4 Pointer Arithmetic 434
Pointer Initialization 438
10.5 Passing and Using Array Addresses 439
Advanced Pointer Notation 442
10.6 Passing and Returning File Names 446
10.7 Common Programming Errors 449
10.8 Chapter Summary 451

CHAPTER ELEVEN Character Strings 453

11.1 String Fundamentals 454
String Input and Output 454
String Processing 456
Character-by-Character Input 458
11.2 Pointers and Library Functions 461
Library Functions 466
11.3 String Definitions and Pointer Arrays 468
Pointer Arrays 471
11.4 Formatting Strings 475
In-Memory String Conversions 476
Format Strings 477
11.5 Common Programming Errors 478
11.6 Chapter Summary 479

CHAPTER TWELVE Structures 480

12.1 Single Structures 481
12.2 Arrays of Structures 487
12.3 Passing and Returning Structures 491
Returning Structures 495
12.4 Linked Lists 498

12.5 Dynamic Storage Allocation 505
12.6 Unions 512
12.7 Common Programming Errors 514
12.8 Chapter Summary 515

CHAPTER THIRTEEN Bit Operations 516

13.1 The AND Operator 517
13.2 The Inclusive OR Operator 520
13.3 The Exclusive OR Operator 522
13.4 The Complement Operator 524
13.5 Different-Size Data Items 525
13.6 The Shift Operators 526
13.7 Chapter Summary 529

CHAPTER FOURTEEN Additional Capabilities 530

14.1 Expressions Revisited 531
Conditional Expressions 533
14.2 User-Specified Data Types 535
Enumerated Data Types 535
The `typedef` Statement 538
14.3 Defining Macros 539
14.4 Command Line Arguments 543
14.5 The `goto` Statement 548
14.6 Chapter Summary 549

Appendixes 551

A. Program Entry, Compilation, and Execution under the DOS, UNIX, VAX-VMS, and PRIME Operating Systems 552
B. Using Borland's Turbo C Compiler 559
C. Using Microsoft's Quick C Compiler 563
D. Living with an Old Compiler 567
E. Operator Precedence Table 574
F. ASCII Character Codes 576
G. Floating Point Number Storage 577

Index 637

Getting Started

Chapter One

1.1 Introduction to Programming
1.2 Introduction to Modularity
1.3 The `printf( )` Function
1.4 Programming Style
1.5 Common Programming Errors
1.6 Chapter Summary
1.7 Enrichment Study: Computer Hardware and Storage

1.1 Introduction to Programming

A computer is a machine. Like other machines, such as an automobile or lawn mower, it must be turned on and then driven, or controlled, to do the task it was meant to do. In an automobile, for example, control is provided by the driver, who sits inside and directs the car. In a computer, the driver is a set of instructions called a program. More formally, a *computer program* is a sequence of instructions that is used to operate a computer to produce a specific result. *Programming* is the process of writing these instructions in a language that the computer can respond to and that other programmers can understand. The set of instructions that can be used to construct a program is called a *programming language.*

On a fundamental level, all computer programs do the same thing (Figure 1–1). They direct a computer to accept data (input), to manipulate the data (process), and to produce reports (output). This implies that all computer programming languages must provide essentially the same capabilities for performing these operations. These capabilities are provided either as specific instruction types or as "prepackaged" groups of instructions that can be called to do specific tasks. In C, the "prepackaged" groups of instructions are called *library functions.* Table 1–1 lists the fundamental set of instructions and library functions provided by FORTRAN, BASIC, COBOL, Pascal, and C for performing input, processing, and output tasks.

If all programming languages provide essentially the same features, why are there so many of them? The answer is that there are vast differences in the types of input data, calculations needed, and output reports required by various applications. For example, scientific and engineering applications require high-precision numerical outputs, accurate to many decimal places. In addition, these applications typically use many algebraic or trigonometric formulas to produce their results. For example, the determination of a rocket's reentry point, as illustrated in Figure 1–2, requires a trigonometric formula and a high degree of numerical accuracy. For such applications, the FORTRAN programming language, with its algebralike instructions, was initially developed. FORTRAN, whose name is an acronym derived from FORmula TRANslation, was introduced in 1957.

Business applications usually deal in whole numbers, representing inventory quantities, for example, or dollars and cents data accurate to only two decimal places. These applications require simpler mathematical calculations than are needed for scientific applications. The outputs required from business programs frequently consist of reports containing extensive columns of neatly formatted dollars and cents numbers and totals (see Figure 1–3). For these applications the

FIGURE 1–1 All Programs Perform the Same Operations

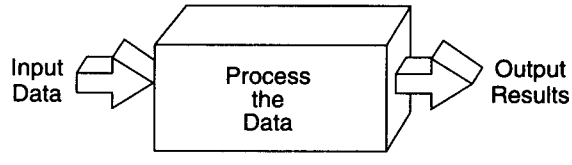


TABLE 1–1 Fundamental Programming Language Instruction Summary

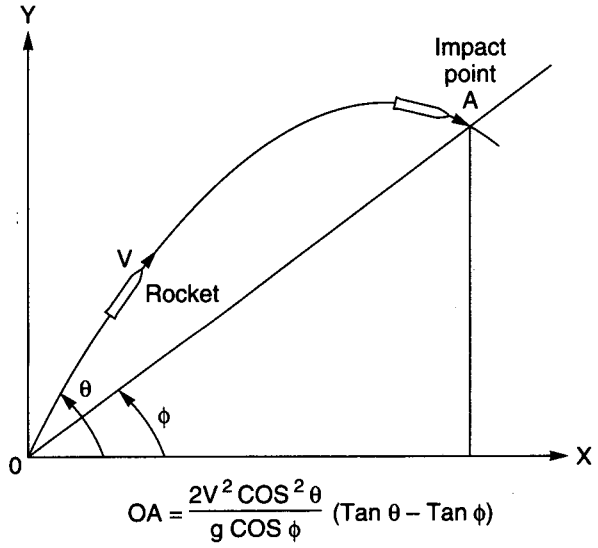
Operation	FORTRAN	BASIC	COBOL	Pascal	C
Input (Get the data)	READ	INPUT READ/DATA	READ ACCEPT	READ READLN	getchar() gets() scanf() sscanf() fscanf()
Processing (Use the data)	= IF/ELSE DO	LET IF/ELSE FOR	COMPUTE IF/ELSE PERFORM	:= IF/ELSE FOR WHILE REPEAT	= if/else for while do
Output (Display the data)	WRITE PRINT	PRINT PRINT/ USING	WRITE DISPLAY	WRITE WRITELN	putchar() puts() printf() sprintf() fprintf()

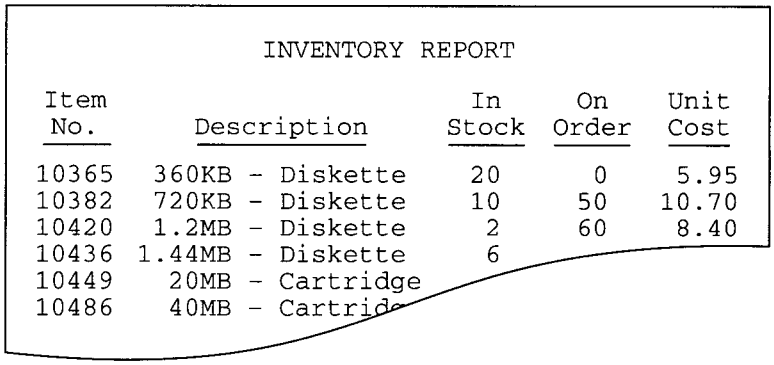
COBOL programming language, with its picture output formats, is an ideal language. COBOL, which was commercially introduced in the 1960s, stands for COmmon Business Oriented Language.

Teaching programming to students has its own set of requirements. Here, a relatively straightforward, easy-to-understand language is needed that does not require detailed knowledge of a specific application. Both the BASIC and Pascal programming languages were developed for this purpose. BASIC, which stands for Beginners All-purpose Symbolic Instruction Code, was developed in the 1960s at Dartmouth College. BASIC is ideal for creating small, easily developed, interactive programs.

Pascal was developed in the late 1970s to provide students with a firmer foundation in modular and structured programming than could be provided by

FIGURE 1–2 FORTRAN Was Developed for Scientific and Engineering Applications





INVENTORY REPORT

Item No.	Description	In Stock	On Order	Unit Cost
10365	360KB - Diskette	20	0	5.95
10382	720KB - Diskette	10	50	10.70
10420	1.2MB - Diskette	2	60	8.40
10436	1.44MB - Diskette	6		
10449	20MB - Cartridge			
10486	40MB - Cartrid			

FIGURE 1–3 COBOL Is Ideal for Many Business Applications

BASIC. Modular programs consist of many small subprograms, each of which performs a clearly defined and specific task that can be tested and modified without disturbing other sections of the program. The name Pascal is not an acronym, like the words FORTRAN, COBOL, and BASIC; the language is named after the seventeenth-century mathematician Blaise Pascal. The Pascal language is so rigidly structured, however, that there are no escapes from the structured modules when such escapes would be useful. This is unacceptable for real-world projects, and is one of the reasons that Pascal has not become widely accepted in the scientific, engineering, and business fields. The design philosophy called structured programming that led to the development of Pascal does have relevance to us as C programmers, however. Using a structured programming approach results in readable, reliable, and maintainable programs. We introduce the elements of this program design philosophy in the next section and continue to expand upon it and use it throughout the text.

The C language was initially developed in the 1970s by Ken Thompson, Dennis Ritchie, and Brian Kernighan at AT&T Bell Laboratories. C evolved from a language called B, which was itself developed from the language BCPL. C has an extensive set of capabilities and is a true general-purpose programming language. As such, it can be used for creating simple, interactive programs or highly sophisticated and complex engineering and scientific programs, within the context of a truly structured language. An indication of C's richness of library and instruction capabilities is clearly evident in Table 1–1. Not only does C initially provide many "tools" to build programs with, but as we shall see, it also provides the programmer with the ability to easily create new "tools" to add to the existing library routines. For this reason C has become known as the professional programmer's language.

Algorithms

Before a program is written, the programmer must have a clear understanding of what the desired result is and how the proposed program is to produce it. In this regard, it is useful to realize that a computer program describes a computational procedure.

In computer science, a computational procedure is called an algorithm. More specifically, an *algorithm* is defined as a step-by-step sequence of instructions that describes how a computation is to be performed. In essence, an algorithm answers the question, "What method will you use to solve this computational problem?" Only after we clearly understand the algorithm and know the specific

steps required to produce the desired result can we write the program. Seen in this light, programming is the translation of the selected algorithm into a language that the computer can use.

To illustrate an algorithm, we shall consider a simple requirement. Assume that a program must calculate the sum of all whole numbers from 1 through 100. Figure 1–4 illustrates three methods we could use to find the required sum. Each method constitutes an algorithm.

Clearly, most people would not bother to list the possible alternatives in a detailed step-by-step manner, as is done in Figure 1–4, and then select one of the algorithms to solve the problem. But then, most people do not think algorithmically; they tend to think intuitively. For example, if you had to change a flat tire on your car, you would not think of all the steps required—you would simply change the tire or call someone else to do the job. This is an example of intuitive thinking.

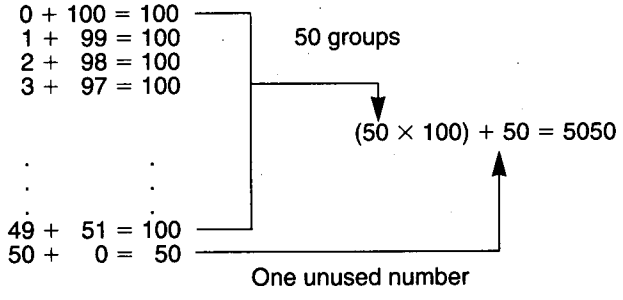
Unfortunately, computers do not respond to intuitive commands. A general

FIGURE 1–4 Summing the Numbers 1 through 100

Method 1. *Columns:* Arrange the numbers from 1 to 100 in a column and add them:

$$\begin{array}{r} 1 \\ 2 \\ 3 \\ 4 \\ \vdots \\ 98 \\ 99 \\ +100 \\ \hline 5050 \end{array}$$

Method 2. *Groups:* Arrange the numbers in convenient groups that sum to 100. Multiply the number of groups by 100 and add in any unused numbers:



Method 3. *Formula:* Use the Formula

$$\text{Sum} = \frac{n(a + b)}{2}$$

where

n = number of terms to be added (100)
a = first number to be added (1)
b = last number to be added (100)

$$\text{Sum} = \frac{100\ (1 + 100)}{2} = 5050$$

statement such as "add the numbers from 1 to 100" means nothing to a computer, because the computer can only respond to algorithmic commands written in an acceptable language such as C. To program a computer successfully, you must clearly understand this difference between algorithmic and intuitive commands. A computer is an "algorithm-responding" machine; it is not an "intuitive-responding" machine. You cannot tell a computer to change a tire or to add the numbers from 1 through 100. Instead, you must give the computer a detailed step-by-step set of instructions that, collectively, forms an algorithm. For example, the set of instructions

Set n equal to 100
Set a = 1
Set b equal to 100
Calculate sum $= \frac{n(a+b)}{2}$
Print the sum

forms a detailed method, or algorithm, for determining the sum of the numbers from 1 through 100. Notice that these instructions are not a computer program. Unlike a program, which must be written in a language the computer can respond to, an algorithm can be written or described in various ways. When English-like phrases are used to describe the algorithm (the processing steps), as in this example, the description is called *pseudocode.* When mathematical equations are used, the description is called a *formula.* When pictures that employ specifically defined shapes are used, the description is referred to as a *flowchart.* A flowchart provides a pictorial representation of the algorithm using the symbols shown in Figure 1–5. Figure 1–6 illustrates the use of these symbols in depicting an algorithm for determining the average of three numbers.

Because flowcharts are cumbersome to revise, the use of pseudocode to express the logic of an algorithm has gained increasing acceptance in recent years among programmers. Unlike flowcharts, where standard symbols are defined, there are no standard rules for constructing pseudocode. In describing an algorithm using pseudocode, any short English phrases may be used. For example, acceptable pseudocode for describing the steps needed to compute the average of three numbers is:

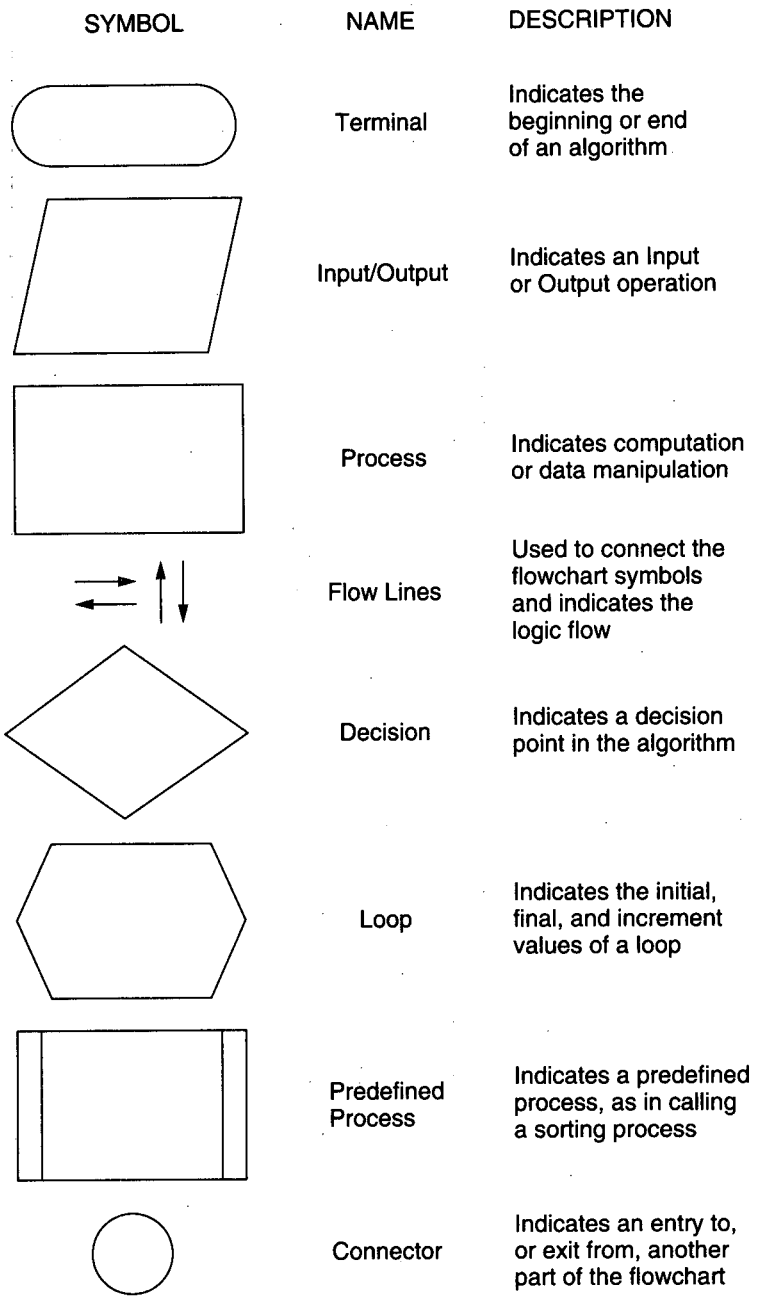
Input the three numbers into the computer
Calculate the average by adding the numbers and dividing the sum by three
Display the average

Only after an algorithm has been selected and the programmer understands the steps required can the algorithm be written using computer-language statements. When computer-language statements are used to describe the algorithm, the description is called a *computer program.*

From Algorithms to Programs

After an algorithm has been selected, it must be converted into a form that can be used by a computer. The conversion of an algorithm into a computer program, using a language such as C, is called *coding* the algorithm (see Figure 1–7). Much of the remainder of this text is devoted to showing you how to develop algorithms and express those algorithms in C.

FIGURE 1–5 Flowchart Symbols



Program Translation

Once a program is written in C it still cannot be executed on a computer without further translation. This is because the internal language of all computers consists of a series of 1s and 0s, called the computer's *machine language.* To generate a machine-language program that can be executed by the computer requires that the C program, which is referred to as a *source program,* be translated into the computer's machine language (see Figure 1–8).

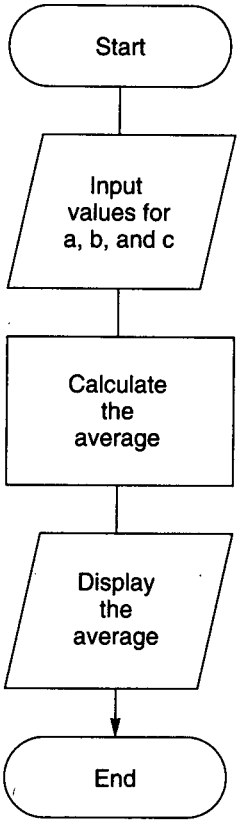


FIGURE 1–6 Flowchart for Calculating the Average of Three Numbers

The translation into machine language can be accomplished in two ways. When each statement in the source program is translated individually and executed immediately, the programming language used is called an *interpreted language*, and the program doing the translation is called an *interpreter*.

When all of the statements in a source program are translated before any one statement is executed, the programming language used is called a *compiled language*. In this case, the program doing the translation is called a *compiler*. C is a compiled language. Here, the source program is translated as a unit into machine

FIGURE 1–7 Coding an Algorithm

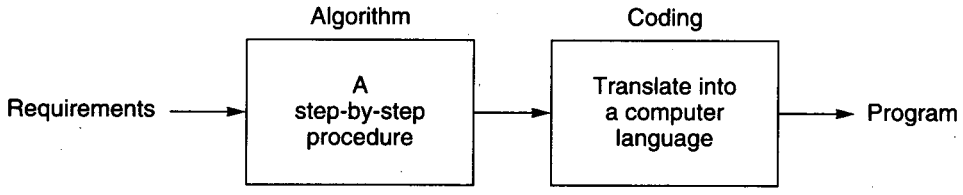


FIGURE 1–8 Source Programs Must be Translated

language. The machine-language version of the original source program is a separate entity; it is called the *object program.* (See Appendix A for a complete description of entering, compiling, and running a C program.)

Exercises 1.1

1. Define the terms:
   a. computer program
   b. programming
   c. programming language
   d. algorithm
   e. pseudocode
   f. flowchart
   g. source program
   h. object program
   i. compiler
   j. interpreter
2. Determine a step-by-step procedure (list the steps) to do these tasks:

(Note: There is no one single correct answer for each of these tasks. This exercise is designed to give you practice in converting intuitive commands into equivalent algorithms and making the shift between the thought processes involved in the two types of thinking.)

   a. Fix a flat tire.
   b. Make a telephone call.
   c. Go to the store and purchase a loaf of bread.
   d. Roast a turkey.

3. Determine and write an algorithm (list the steps) to interchange the contents of two cups of liquid. Assume that a third cup is available to hold the contents of either cup temporarily. Each cup should be rinsed before any new liquid is poured into it.
4. Write a detailed set of instructions in English to calculate the dollar amount of money in a piggybank that contains h half-dollars, q quarters, n nickels, d dimes, and p pennies.
5. Write a set of detailed, step-by-step instructions in English to find the smallest number in a group of three integer numbers.
6 *a.* Write a set of detailed, step-by-step instructions in English to calculate the change remaining from a dollar after a purchase is made. Assume that the cost of the goods purchased is less than a dollar. The change received should consist of the smallest number of coins possible.
   b. Repeat Exercise 6a, but assume the change is to be given only in pennies.
7 *a.* Write an algorithm to locate the first occurrence of the name JONES in a list of names arranged in random order.
   b. Discuss how you could improve your algorithm for Exercise 7a if the list of names was arranged in alphabetical order.
8. Write an algorithm to determine the total occurrences of the letter *e* in any sentence.
9. Determine and write an algorithm to sort four numbers into ascending (from lowest to highest) order.

1.2 Introduction to Modularity

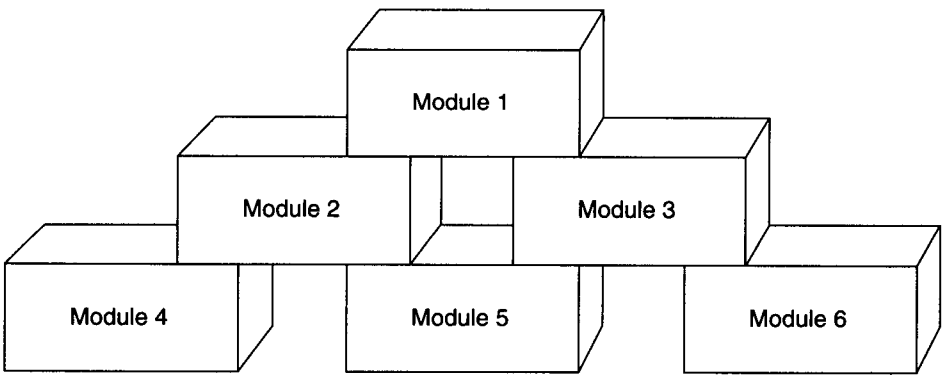
A well-designed program is constructed using a design philosophy similar to that used in constructing a well-designed building. It doesn't just happen, but depends on careful planning and execution for the final design to accomplish its intended purpose. Just as an integral part of the design of a building is its structure, the same is true for a program.

In programming, the term *structure* has two interrelated meanings. The first meaning refers to the program's overall construction, which is the topic of this section. The second meaning refers to the form used to carry out the individual tasks within the program, which is the topic of Chapters 4 and 5. In relation to the first meaning, programs whose structure consists of interrelated segments arranged in a logical and easily understandable order to form an integrated and complete unit are referred to as *modular programs* (Figure 1–9). Not surprisingly, it has been found that modular programs are noticeably easier to develop, correct, and modify than programs constructed otherwise. In general programming terminology, the smaller segments used to construct a modular program are referred to as *modules.*

Each module is designed and developed to perform a specific task and is really a small subprogram all by itself. A complete C program is constructed by combining as many modules as necessary to produce the desired result. The advantage to this modular construction is that the overall design of the program can be developed before any single module is written. Once the requirements for each module are finalized, the modules can be programmed and integrated within the overall program as they are completed.

Since a module is really a small subprogram, each module must be able to do what is required of all programs: receive data, process the data, and produce a result (see Figure 1–10). Unlike a larger program, however, a module performs operations that are very limited in nature. Modules are meant to handle at most one or two functions required by the complete program. Since each module is designed to perform specific functions, the modules themselves are called *functions* in C.

FIGURE 1–9 A Well-Designed Program Is Built Using Modules



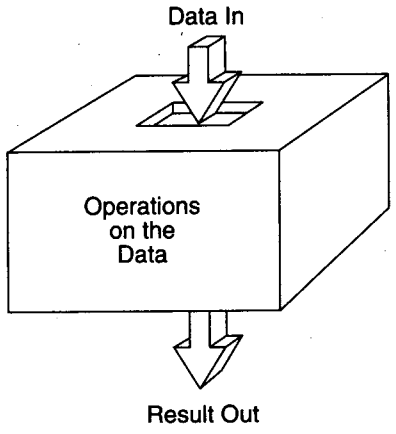


FIGURE 1–10 A Module Must Accept Data, Process the Data, and Produce a Result

Functions

It is useful to think of a function as a small machine that transforms the data it receives into a finished product. For example, Figure 1–11 illustrates a function that accepts two numbers as inputs and multiplies the two numbers to produce one output.

One important requirement for designing a good function is to give it a name that conveys to the reader some idea about what the function does. Function names can be made up of any combination of letters, digits, or underscores (_) selected according to the following rules:

1. The function name must begin with a letter or underscore.
2. Only letters, digits, or underscores may follow the initial letter. Blank spaces are not allowed; use the underscore to separate words in a name consisting of multiple words.

FIGURE 1–11 A Multiplying Function

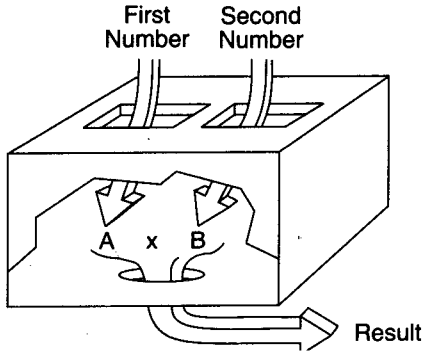


TABLE 1–2 Keywords

auto	do	for	return	typedef
break	double	goto	short	union
case	else	if	sizeof	unsigned
char	enum	int	static	void
continue	extern	long	struct	while
default	float	register	switch	

3. A function name cannot be one of the keywords listed in Table 1–2. (A *keyword* is a word that is set aside by the language for a special purpose and should only be used in a specified manner.[1])
4. The maximum number of characters in a function name is computer dependent. However, all systems recognize at least eight characters. (The American National Standards Institute (ANSI) standard requires recognition of at least 31 characters)
5. All function names are followed by a single set of parentheses.

Examples of valid C function names, including the required parentheses, are:

```
deg_to_rad( )   intersct( )   add_nums( )   slope( )
bessel1( )      mult_two( )   find_max( )   density( )
```

Examples of invalid function names are:

```
1AB3( )     (begins with a number, which violates Rule 1)
E*6( )      (contains a special character, which violates Rule 2)
while( )    (this is a keyword, which violates Rule 3)
```

Besides conforming to the rules for naming functions, a good function name should also be a mnemonic. A *mnemonic* is a word or name designed as a memory aid. For example, the function name `deg_to_rad( )` is a mnemonic if it is the name of a function that converts degrees to radians. Here, the name itself helps to identify what the function does.

Examples of valid function names that are not mnemonics are:

```
easy( )   c3po( )   r2d2( )   theforce( )   mike( )
```

Nonmnemonic function names should not be used because they convey no information about what the function does.

[1] Keywords in C are also reserved words, which means they can only be used for a specified purpose.

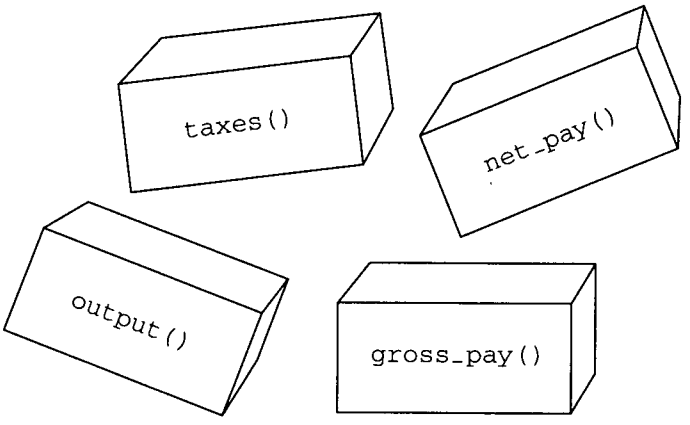


FIGURE 1–12 We Need Some Order Here!

Notice that all function names have been typed in lowercase letters. This is traditional in C, although it is not absolutely necessary. Uppercase letters are usually reserved for named constants, a topic covered in Chapter 3. Note that C is a *case-sensitive* language. This means that the compiler distinguishes between uppercase and lowercase letters. Thus, in C, the names `TOTAL`, `total`, and `Total` represent three distinct and different names. For this reason, we will type all names in the same case, which traditionally is lowercase in C.

The `main( )` Function

Once functions have been named, we need a way to combine them into a complete program (see Figure 1–12). Notice that we have not yet described the actual writing of the functions. One of the nice features of C is that we can plan a program by first deciding what functions are needed and how they are to be linked together. Then we can write each function to perform the task it is required to do.

To provide for the orderly placement and execution of functions, each C program must have one function called `main( )`. The `main( )` function is sometimes referred to as a driver function, because it tells the other functions the sequence in which they are to operate (see Figure 1–13).

Figure 1–14 illustrates a completed `main( )` function. The word `main` identifies the start of each C program. The braces, `{` and `}`, determine the beginning and end of the function body and enclose the statements making up the function. The statements inside the braces determine what the function does. Each statement inside the function must end with a semicolon (`;`).

The `main( )` function illustrated in Figure 1–14 consists of four statements. In this case, each statement is a command to execute another function. First the `gross_pay( )` function is called for execution. When `gross_pay( )` is finished, the `taxes( )` function is called. After the `taxes( )` function is completed, the `net_pay( )` function is called. Finally, the `output( )` function is executed. Although the functions `gross_pay( )`, `taxes( )`, `net_pay( )`, and `output( )` must still be written, the `main( )` function is completed. After the four other functions are written, the program, consisting of `main( )`, `gross_pay( )`, `taxes( )`, `net_pay( )`, and `output( )`, is complete.

You will be naming and writing many of your own C functions. In fact, the rest of this book is primarily about the statements required to construct useful

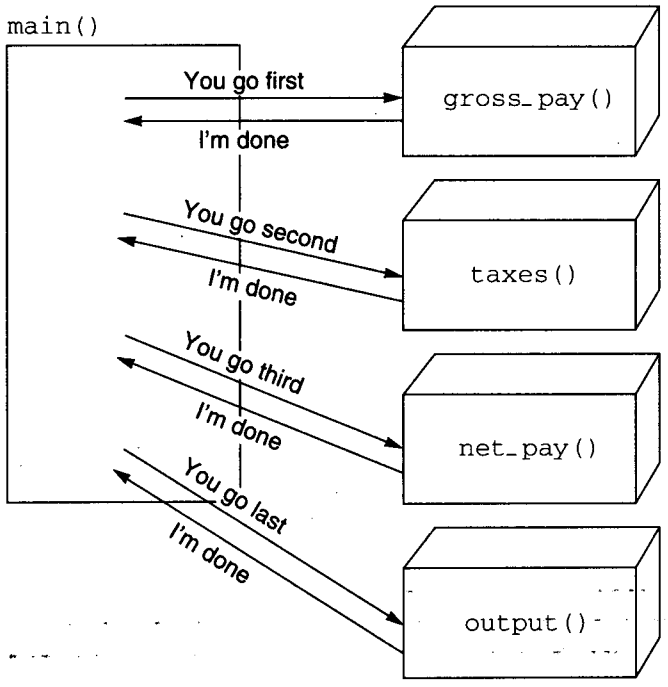


FIGURE 1–13 The **main( )** Function Controls All Other Functions

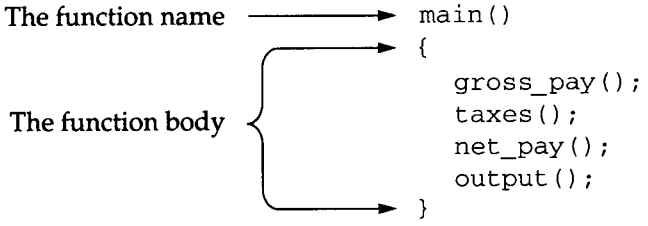


FIGURE 1–14 A Sample `main( )` Function

functions and about how to combine the functions to form useful programs. Fortunately, however, many useful functions have already been written for us. In the next section we will use one of these functions to create our first working C program.

Exercises 1.2

1. State whether the following are valid function names. If they are valid, state whether they are mnemonic names (recall that a mnemonic function name conveys some idea about what the function's purpose). If they are invalid names, state why.

m1234( )	new_bal( )	abcd( )	A12345( )	1A2345( )
power( )	abs_val( )	mass( )	do( )	while( )
add_5( )	taxes( )	net_pay( )	12345( )	int( )
cosine( )	a2b3c4d5( )	salestax( )	amount( )	$sine( )

2. Assume that the following functions have been written:

```
old_bal( ), sold( ), new_bal( ), report( )
```

   a. Write a C program that calls these functions in the order that they are listed.
   b. From the functions' names, what do you think each function might do?

3. Assume that the following functions have been written:

```
input( ), salestax( ), balance( ), calcbill( )
```

   a. Write a C program that calls these functions in the order that they are listed.
   b. From the functions' names, what do you think each function might do?

4. Create valid mnemonic names for functions that do the following:
   a. Find the average of a set of numbers.
   b. Find the area of a rectangle.
   c. Find the value of a polynomial.
   d. Find the density of a steel door.
   e. Find the maximum value of a set of numbers.
   f. Sort a set of numbers from lowest to highest.

Project Structuring Exercises

Most projects, both programming and nonprogramming, can be structured into smaller subtasks or units of activity. These smaller subtasks can often be delegated to different people so that when all the tasks are finished and integrated, the project or program is completed. For exercises 5 through 10, determine a set of subtasks that, taken together, complete the required task.

(*Note:* The purpose of these exercises is to have you consider the different ways that complex tasks can be structured. Although there is no one correct solution to these exercises, there are incorrect solutions and solutions that are better than others. An incorrect solution is one that does not complete the task correctly. One solution is better than another if it more clearly or easily identifies what must be done or does it more efficiently.

5. You are given the task of wiring and installing lights in the attic of your house. Determine a set of subtasks that, taken together, will accomplish this. (*Hint:* The first subtask would be to determine the placement of the light fixtures.)

6. You are given the job of preparing a complete meal for five people next weekend. Determine a set of subtasks that, taken together, accomplish this. (*Hint:* One subtask, not necessarily the first one, would be to buy the food.)

7. You are a sophomore in college and are planning to go to graduate school for a master's degree in electrical engineering after you graduate. List a set of major objectives that you must fulfill to meet this goal. (*Hint:* One objective is "Take the right courses.")

8. You are given the job of planting a vegetable garden. Determine a set of subtasks that accomplish this. (*Hint:* One such subtask would be to plan the layout of the garden.)

9. You are responsible for planning and arranging the family camping trip this summer. List a set of subtasks that, taken together, accomplish this objective successfully. (*Hint:* One subtask would be to select the camp site.)

10a. A national electrical supply distribution company desires a computer system to prepare its customer invoices. The system must, of course, be capable of creating each day's invoices. Additionally, the company wants the capability to retrieve and output a printed report of all invoices that meet certain criteria; for example, all invoices sent in a particular month with a net value of more than a given dollar amount, all invoices sent in a year to a particular client, or all invoices sent to firms in a particular state. Determine three or four major program units into which the system could be separated. (*Hint:* One program unit is "Prepare Invoices" to create each day's invoices.)
b. Suppose someone enters incorrect data for a particular invoice, which is discovered after the data has been entered and stored by the system. What program unit is needed to take care of correcting this problem? Discuss why such a program unit might or might not be required by most business systems.
c. Assume a program unit exists that allows a user to alter or change data that have been incorrectly entered and stored. Discuss the need for including an "audit trail" that would allow for a later reconstruction of the changes made, when they were made, and who made them.

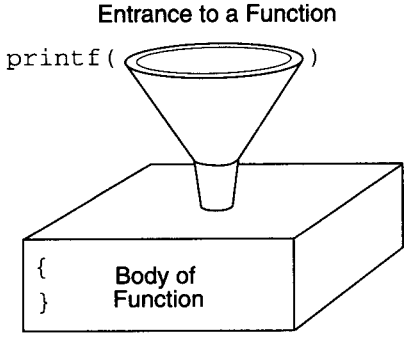
1.3 The `printf( )` Function

One of the most popular and useful prewritten functions in C is named `printf( )`. This function, as its name suggests, is a print function that sends data given to it to the standard system display device. For most systems this display device is a video screen. This function prints out whatever is given to it. For example, if the message `Hello there world!` is given to `printf( )`, this message is displayed on your terminal by the `printf( )` function. Inputting data or messages to a function is called passing data to the function. The message `Hello there world!` is passed to the `printf( )` function by simply putting the message inside the parentheses in the function's name (see Figure 1–15).

FIGURE 1–15 Passing a Message to `printf( )`

```
printf("Hello there world!");
```

FIGURE 1–16 Passing a Message to `printf( )`



The purpose of the parentheses in all function names is to provide a funnel through which information can be passed to the function (see Figure 1–16). The items that are passed to the function through the parentheses are called arguments of the function.

Now let's put all this together into a working C program that can be run on your computer. Consider Program 1-1.

Program 1-1

```
#include <stdio.h>
main( )
{
  printf("Hello there world!");
}
```

The first line of the program:

```
#include <stdio.h>
```

is a preprocessor command. Preprocessor commands begin with a pound sign, #, and perform some action before the compiler translates the source programinto machine code. Specifically, the `#include` preprocessor command causes the contents of the named file, in this case `stdio.h`, to be inserted where the `#include` command appears. The file `stdio.h` is referred to as a *header file* because it is placed at the top, or head, of a C program using the `#include` command. In particular, the `stdio.h` file provides a proper interface to the `printf( )` function, and must be included in all programs using `printf( )`.[2]

As indicated in Program 1-1, preprocessor commands do not end with a semicolon.

Following the preprocessor command is the start of the program's `main( )` function. The `main( )` function itself has only one statement. Remember that statements end with a semicolon (;). The statement in `main( )` calls the function `printf( )` and passes one argument to it. The argument is the message `Hello there world!`

Since `printf( )` is a prewritten function, we do not have to write it; it is available for use just by calling it correctly. Like all C functions, `printf( )` was written to do a specific task, which is to print results. It is a very versatile function that can print results in many different forms. When a message is passed to `printf( )`, the function sees to it that the message is correctly printed on your terminal, as shown in Figure 1–17.

Messages are called strings in C because they consist of a string of characters made up of letters, numbers, and special characters. The beginning and end of a

[2] Use of this statement is required under ANSI-standard C. In non-ANSI C the `#include` statement for stdio.h can usually be omitted.

```
Hello there world!
```

FIGURE 1-17 The Output from Program 1-1

string of characters is marked by using double quotes (`"message in here"`) around the string. Thus, to pass a message to `printf( )`, the string of characters making up the message must be enclosed in double quotes, as we have done in Program 1-1.

Let us write another program to illustrate `printf( )`'s versatility. Read Program 1-2 to determine what it does.

Program 1-2

```
#include <stdio.h>
main( )
{
  printf("Computers, computers everywhere");
  printf("\n   as far as I can C");

}
```

When Program 1-2 is run, the following is displayed:

```
Computers, computers everywhere
   as far as I can C
```

You might be wondering why the `\n` did not appear in the output. The two characters `\` and `n`, when used together, are called a newline escape sequence. They tell `printf( )` to start on a new line. In C, the backslash (`\`) character provides an "escape" from the normal interpretation of the character following it by altering the meaning of the next character. If the backslash was omitted from the second `printf( )` call in Program 1-2, the `n` would be printed as the letter n and the program would print out:

```
Computers, computers everywheren   as far as I can C
```

Newline escape sequences can be placed anywhere within the message passed to `printf( )`. See if you can determine what the following program prints:

```
#include
main( )
{
  printf("Computers everywhere\n as far as\n\nI can see");
}
```

The output for this program is:

```
Computers everywhere
 as far as

I can see
```

Exercises 1.3

1 a. Using the `printf( )` function, write a C program that prints your name on one line, your street address on a second line, and your city, state, and zip code on the third line.
b. Compile and run the program you have written for Exercise 1a.

(Note: To do this you must understand the procedures for entering, compiling, and running a C program on the particular computer you are using.)

2 a. Write a C program to print out the following:

```
THE COSECANT OF AN ANGLE
  IS EQUAL TO ONE OVER
     THE SINE OF THE SAME ANGLE.
```

b. Compile and run the program you have written for Exercise 2a.

3 a. How many `printf( )` statements should be used to display the following:

```
DEGREES      RADIANS
      0      0.0000
     90      1.5708
    180      3.1416
    270      4.7124
    360      6.2832
```

b. What is the minimum number of `printf( )` statements that could be used to print the table in Exercise 3a? Why would you not write a program using the minimum number of `printf( )` function calls?
c. Write a complete C program to produce the output illustrated in Exercise 3a.
d. Run the program you have written for Exercise 3c on a computer.

4. In response to a newline escape sequence, `printf( )` positions the next displayed character at the beginning of a new line. This positioning of the next character actually represents two distinct operations. What are they?

5 a. Most computer operating systems provide the capability for redirecting the output produced by `printf( )` either to a printer or directly to a floppy or hard disk file. If your computer supports output redirection, run the program written for Exercise 2a using this feature. Have your program's display redirected to a file named `poem`.
b. If your computer supports output redirection to a printer, run the program written for Exercise 2a using this feature.

1.4 Programming Style

The word `main` in a C program tells the computer where the program starts. Since a program can have only one starting point, every C language program must contain one and only one `main( )` function. As we have seen, all of the statements that make up the `main( )` function are then included within the braces `{ }` following the function name. Although the `main( )` function must be present in every C program, C does not require that the word `main`, the parentheses `( )`, or the braces `{ }` be placed in any particular form. The form used in the last section

```
main( )
{
  program statements in here;
}
```

was chosen strictly for clarity and ease in reading the program.[3] For example, the following general form of a `main( )` function would also work:

```
main
(
) { first statement;second statement;
third statement;fourth
statement;}
```

Notice that more than one statement can be put on a line, or one statement can be written across lines. Except for messages contained within double quotes, function names, and reserved words, C ignores all white space (white space refers to any combination of one or more blank spaces, tabs, or new lines). For example, changing the white space in Program 1-1 and making sure not to split the message `Hello there world!` or the function names `printf` and `main` across two lines results in the following valid program:

```
#include <stdio.h>
main
(
){printf
("Hello there world!"
);}
```

Although this version of `main( )` does work, it is an example of extremely poor programming style. It is difficult to read and understand. For readability, the `main( )` function should always be written in standard form as:

[3] If one of the program statements was a call to `printf( )`, the `#include <stdio.h>` preprocessor command would have to be used.

```
main()
{
  program statements in here;
}
```

In this standard form the function name starts in column 1 and is placed with the required parentheses on a line by itself. The opening brace of the function body follows on the next line and is placed under the first letter of the function name. Similarly, the closing function brace is placed by itself in column 1 as the last line of the function. This structure serves to highlight the function as a single unit.

Within the function itself, all program statements are typically indented at least two spaces, although indentation is not required. Indentation is another sign of good programming practice, especially if the same indentation is used to align similar groups of statements. Review Program 1-2 to see that the same indentation was used for both `printf( )` function calls.

As you progress in your understanding and mastery of C, you will develop your own indentation standards. Just keep in mind that the final form of your programs should be consistent and should always serve as an aid to the reading and understanding of your programs.

Comments

Comments are explanatory remarks made within a program. When used carefully, comments can be very helpful in clarifying what the complete program is about, what a specific group of statements is meant to accomplish, or what one line is intended to do.

Any line of text bounded by asterisks and enclosed within slashes (/) is a comment. For example,

```
/* this is a comment */
/* this program prints out a message */
/* this program calculates a square root */
```

are all comment lines. The symbols `/*`, with no white space between them, designate the start of a comment. Similarly, the symbols `*/`, as a single unit with no intervening white space, designate the end of a comment.

Comments can be placed anywhere within a program. They have no effect on program execution. The computer ignores all comments—they are there strictly for the convenience of anyone reading the program.

A comment can be written either on a line by itself or on the same line containing a program statement. Program 1-3 illustrates the use of comments within a program.

Program 1-3

```
#include <stdio.h>
main( )    /* this program prints a message */
{
  printf("Hello there world!"); /* a call to printf( ) */
}
```

The first comment appears on the same line as the function name and describes what the program does. This is generally a good location to include a short comment describing the program's purpose. If more comments are required, they can be placed, one per line, between the function name and the opening brace that encloses the function's statements. If a comment is too long to be contained on one line, it can be continued across two or more lines as illustrated below:

```
/* this comment is used to illustrate a
comment that extends over two lines */
```

Note, however, that under older C compilers this usually results in an error. For these compilers the comment must be split into two or more comments, with each separate comment enclosed within its own comment symbol set `/*  */` as follows:

```
/* this comment is used to illustrate a */
/* comment that is split into two comments */
```

Under no circumstances (may comments be nested (one comment contains another comment within itself). For example,

```
/* this nested comment is /* always */ invalid */
```

Typically, many comments are required when using nonstructured programming languages. These comments are necessary to clarify either the purpose of the program itself or individual sections and lines of code within the program. In C, the program's structure is intended to make the program readable, making the use of excessive comments unnecessary. This is reinforced if both function names and variable names, described in the next chapter, are carefully selected to convey their meaning to anyone reading the program. However, if the purpose of a function or any of its statements is still not clear from its structure, name, or context, include comments where clarification is needed.

Exercises 1.4

1a. Will the following program work?

```
#include <stdio.h>
main( ){printf("Hello there world!");}
```

b. Why is the program given in Exercise 1a not a good program?

2. Rewrite the following programs to conform to good programming practice.

a.

```
#include <stdio.h>
main(
){
printf
(
"The time has come"
);}
```

b.
```
#include <stdio.h>
main
( ){printf("Newark is a city\n");printf(
"In New Jersey\n"); printf
("It is also a city\n"
); printf("In Delaware\n"
);}
```

c.
```
#include <stdio.h>
main(){printf(Reading a program\n");printf(
"is much easier\n"
);printf("if a standard form for main is used\n")
;printf          ("and each statement is written\n");printf(
"on a line by itself\n")
;}
```

d.
```
#include <stdio.h>
main
( ){printf("Every C program"
);printf
("\nmust have one and only one"
);
printf("main function"
);
printf(
"\n the escape sequence of characters")
;printf(
"\nfor a newline can be placed anywhere"
);printf
("\n within the message passed to printf()"
);}
```

3 *a.* When used in a message, the backslash character alters the meaning of the character immediately following it. If we wanted to print the backslash character, we would have to tell `printf( )` to escape from the way it normally interprets the backslash. What character do you think is used to alter the way a single backslash character is interpreted?

b. Using your answer to Exercise 3a, write the escape sequence for printing a backslash.

4 *a.* A *token* of a computer language is any sequence of characters that, as a unit, with no intervening characters or white space, has a unique meaning. Using this definition of a token, determine whether escape sequences, function names, and the reserved words listed in Table 1–1 are tokens of the C language.

b. Discuss whether adding white space to a message alters the message. Discuss whether messages can be considered tokens of C.

c. Using the definition of a token given in Exercise 4a, determine whether the statement "Except for tokens of the language, C ignores all white space" is true.

1.5 Common Programming Errors

Part of learning any programming language is making the elementary mistakes commonly encountered as you begin to use the language. These mistakes tend to be quite frustrating, since each language has its own set of common programming errors waiting for the unwary. The more common errors made when initially programming in C are:

1. Omitting the parentheses after `main`.
2. Omitting or incorrectly typing the opening brace { that signifies the start of a function body.
3. Omitting or incorrectly typing the closing brace } that signifies the end of a function.
4. Misspelling the name of a function; for example, typing `pintf( )` instead of `printf( )`.
5. Forgetting to close the message to `printf( )` with a double quote symbol.
6. Omitting the semicolon at the end of each C statement.
7. Adding a semicolon at the end of the `#include` preprocessor command.
8. Forgetting the `\n` to indicate a new line.
9. Incorrectly typing the letter O for the number zero (0), or vice versa.
10. Incorrectly typing the letter l for the number 1, or vice versa.

The third, fifth, sixth, seventh and eighth errors in this list are initially the most common. It is worthwhile for you to write a program and specifically introduce each of these errors, one at a time, to see what error messages are produced by your compiler. Then, when these error messages appear due to inadvertent errors, you will have had experience in understanding the messages and correcting the errors.

On a more fundamental level, a major programming error made by all beginning programmers is the rush to code and run a program before the programmer fully understands what is required and the algorithms and procedures that will be used to produce the desired result. A symptom of this haste to get a program entered into the computer is the lack of either an outline of the proposed program or a written program itself. Many problems can be caught just by checking a copy of the program, either handwritten or listed from the computer, before it is ever compiled.

1.6 Chapter Summary

1. A C program consists of one or more modules called functions. One of these functions must be called `main( )`. The `main( )` function identifies the starting point of a C program.
2. Many functions, like `printf( )`, are supplied in a standard library of functions provided with each C compiler.
3. When the `printf( )` function is used within a program, the preprocessor command `#include <stdio.h>` must be placed at the top of the program. Preprocessor commands do not end with a semicolon.
4. The simplest C program consists of the single function `main( )`.
5. Following the function name, the body of a function has the general form:

```
{
   All program statements in here;
}
```

6. All C statements must be terminated by a semicolon.
7. The `printf( )` function is used to display text or numerical results. The first argument to `printf( )` can be a message, which is enclosed in double quotes. The text in the message is displayed directly on the screen and may include newline escape sequences for format control.

1.7 Enrichment Study: Computer Hardware and Storage

All computers, from large supercomputers costing millions of dollars to smaller desktop personal computers, must perform a minimum set of functions and provide the capability to:

1. Accept input.
2. Display output.
3. Store information in a logically consistent format (traditionally binary).
4. Perform arithmetic and logic operations on either the input or stored data.
5. Monitor, control, and direct the overall operation and sequencing of the system.

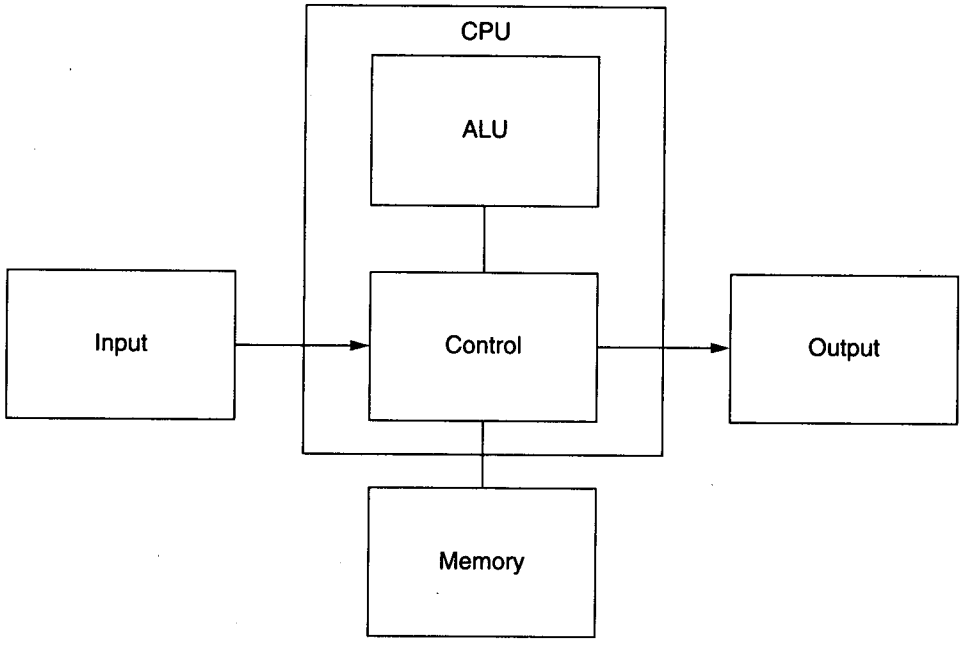
Figure 1–18 illustrates the computer hardware components that support these capabilities. Specifically, this hardware consists of arithmetic and logic, control, memory, and input/output units.

In the first commercially available computers of the 1940s and 1950s, all hardware units were built using relays and vacuum tubes. The resulting computers were extremely large pieces of equipment, capable of making thousands of calculations per second, and costing millions of dollars.

With the commercial introduction of transistors in the 1960s both the size and cost of computer hardware was reduced. The transistor was approximately one-twentieth the size of its vacuum tube counterpart, which allowed manufacturers to combine the arithmetic and logic unit with the control unit into a single new unit. This combined unit is called the central processing unit (CPU). The combination of the ALU and control units into one CPU made sense because a majority of control signals generated by a program are directed to the ALU in response to arithmetic and logic instructions within the program. Combining the ALU with the control unit simplified the interface between these two units and provided improved processing speed.

The mid-1960s saw the introduction of integrated circuits (ICs) that resulted in still another significant reduction in the space required to produce a CPU. Initially, integrated circuits were manufactured with up to 100 transistors on a single 1 cm^2 chip of silicon. Such devices are referred to as small-scale integrated (SSI) circuits. Current versions of these chips contain hundreds of thousands to over a million transistors and are referred to as very large-scale integrated (VLSI) chips.

VLSI chip technology has provided the means of transforming the giant computers of the 1950s into today's desktop personal computers. Each individual unit required to form a computer (CPU, memory, and I/O) is now manufactured on an individual VLSI chip, and the single-chip CPU is referred to as a *micropro-*



Memory Unit: This unit stores information in a logically consistent format. Typically, both instructions and data are stored in memory, usually in separate and distinct areas.

Control Unit: The control unit directs and monitors the overall operation of the computer. It keeps track of where in memory the next instruction resides, issues the signals needed to both read data from and write data to other units in the system, and executes all instructions.

Arithmetic and Logic Unit (ALU): The ALU performs all the arithmetic and logic functions, such as addition, subtraction, comparison, etc., provided by the system.

Input/Output (I/O) Unit: This unit provides access to and from the computer. It is the interface to which peripheral devices such as keyboards, cathode ray screens, printers, and card readers are attached.

FIGURE 1–18 Basic Hardware Units of a Computer

cessor. Figure 1–19 illustrates how these chips are connected internally within current personal computers, such as the IBM-PCs.

Concurrent with the remarkable reduction in computer hardware size has been an equally dramatic decrease in cost and increase in processing speeds. The equivalent computer hardware that cost over a million dollars in 1950 can now be purchased for less than five hundred dollars. If the same reductions occurred in the automobile industry, for example, a Rolls-Royce could now be purchased for ten dollars! The processing speeds of current computers have also increased by a factor of a thousand over their 1950s predecessors, with the computational speeds of current machines being measured in both millions of instructions per second (MIPS) and billions of instructions per second (BIPS).

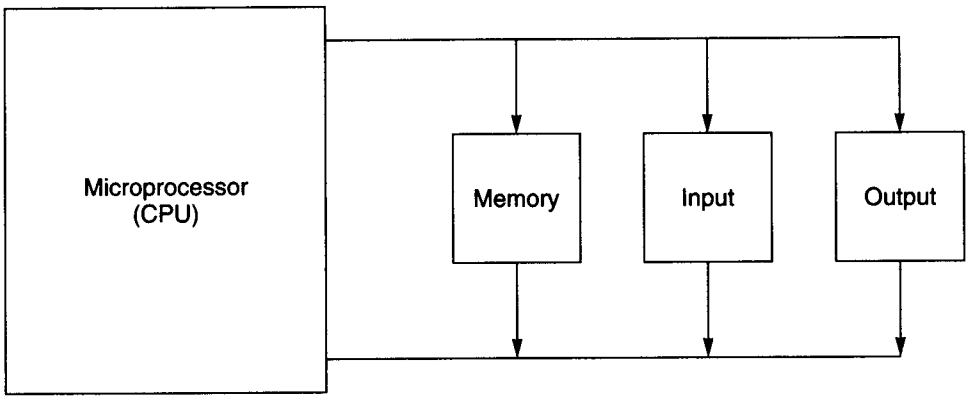


FIGURE 1–19 VLSI Chip Connections for a Desktop Computer

Computer Storage

It would be very convenient if a computer stored numbers and letters inside its memory and arithmetic and logic units the way that people do. The number 126, for example, would then be stored as 126 and the letter A stored as the letter A. Unfortunately, due to the physical components used in building a computer, this is not the case.

The smallest and most basic data item in a computer is called a bit. Physically, a bit is really a switch that can be either open or closed. By convention, the open and closed positions of each switch are represented as a 0 and a 1, respectively.

A single bit that can represent the values 0 and 1, by itself, has limited usefulness. All computers, therefore, group a set number of bits together, both for storage and transmission. The grouping of eight bits to form a larger unit is an almost universal computer standard. Such groups are commonly referred to as bytes. A single byte consisting of eight bits, where each bit is either 0 or 1, can represent any one of 256 distinct patterns. These consist of the pattern 00000000 (all eight switches open) to the pattern 11111111 (all eight switches closed), and all possible combinations of 0s and 1s in between. Each of these patterns can be used to represent either a letter of the alphabet, other single characters, such as a dollar sign, comma, etc., a single digit, or numbers containing more than one digit. The patterns of 0s and 1s used to represent letters, single digits, and other single characters are called *character codes* (two such codes, called the ASCII and EBCDIC codes, are presented in Section 2.1). The patterns used to store numbers are called *number codes*, one of which is presented below.

Two's Complement Numbers

The most common number code for storing integer values inside a computer is called the *two's complement* representation. Using this code, the integer equivalent of any bit pattern, such as 10001101, is easy to determine and can be found for either positive or negative integers with no change in the conversion method. For convenience we will assume byte-sized bit patterns consisting of a set of eight bits each, although the procedure carries directly over to larger size bit patterns.

The easiest way to determine the integer represented by each bit pattern is first to construct a simple device called a value box. Figure 1–20 illustrates such a box for a single byte.

−128	64	32	16	8	4	2	1

FIGURE 1–20 An Eight-Bit Value Box

Mathematically, each value in the box illustrated in Figure 1–20 represents an increasing power of two. Since two's complement numbers must be capable of representing both positive and negative integers, the leftmost position, in addition to having the largest absolute magnitude, also has a negative sign.

Conversion of any binary number, for example 10001101, simply requires inserting the bit pattern in the value box and adding the values having ones under them. Thus, as illustrated in Figure 1–21, the bit pattern 10001101 represents the integer number −115.

The value box can also be used in reverse, to convert a base 10 integer number into its equivalent binary bit pattern. Some conversions, in fact, can be made by inspection. For example, the base 10 number −125 is obtained by adding 3 to −128. Thus, the binary representation of −125 is 10000011, which equals −128 + 2 + 1. Similarly, the two's complement representation of the number 40 is 00101000, which is 32 plus 8.

Although the value box conversion method is deceptively simple, the method is directly related to the underlying mathematical basis of two's complement binary numbers. The original name of the two's complement code was the weighted-sign code, which correlates directly to the value box. As the name weighted sign implies, each bit position has a weight, or value, of two raised to a power and a sign. The signs of all bits except the leftmost bit are positive and the sign of the leftmost bit is negative.

In reviewing the value box, it is evident that any two's complement binary number with a leading 1 represents a negative number, and any bit pattern with a leading 0 represents a positive number. Using the value box it is easy to determine the most positive and negative values capable of being stored. The most negative value that can be stored in a single byte is the decimal number −128, which has the bit pattern 10000000. Any other nonzero bit will simply add a positive amount to the number. Additionally, it is clear that a positive number must have a 0 as its leftmost bit. From this you can see that the largest positive 8-bit two's complement number is 01111111 or 127.

Words and Addresses

One or more bytes may themselves be grouped into larger units called *words*, which facilitate faster and more extensive data access. For example, retrieving a word consisting of four bytes from a computer's memory results in more infor-

FIGURE 1–21 Converting 10001101 to a Base 10 Number

−128	64	32	16	8	4	2	1
1	0	0	0	1	1	0	1

−128 + 0 + 0 + 0 + 8 + 4 + 0 + 1 = −115

mation than that obtained by retrieving a word consisting of a single byte. Such a retrieval is also considerably faster than four individual byte retrievals. This increase in speed and capacity, however, is achieved by an increase in the computer's cost and complexity.

Early personal computers, such as the Apple IIe and Commodore machines, internally stored and transmitted words consisting of single bytes. AT&T 6300 and IBM-PC/XTs use word sizes consisting of two bytes, while Digital Equipment, Data General, and Prime minicomputers store and process words consisting of four bytes each. Supercomputers, such as the CRAY-1 and Control Data 7000, have 6- and 8-byte words, respectively.

The arrangement of words in a computer's memory can be compared to the arrangement of suites in a very large hotel, where each suite is made up of rooms of the same size. Just as each suite has a unique room number to locate and identify it, each word has a unique numeric address. In computers that allow each byte to be individually accessed, each byte has its own address. Like room numbers, word and byte addresses are always positive, whole numbers that are used for location and identification purposes. Also, like hotel rooms with connecting doors for forming larger suites, words can be combined to form larger units for the accommodation of different size data types.

Data Types and Operations

Chapter Two

2.1 Data Constants and Arithmetic Operations
2.2 Variables and Declaration Statements
2.3 Assignment Statements
2.4 Formatted Output
2.5 Top-Down Program Development
2.6 Applications
2.7 Common Programming Errors
2.8 Chapter Summary
2.9 Enrichment Study: Variable Addresses and Memory Storage

2.1 Data Constants and Arithmetic Operations

C recognizes four basic types of data: integers, floating point (single precision) numbers, double precision numbers, and character data. These data types, expressed as constants, are described below.

Integer Constants

An *integer constant* in C is any positive or negative number without a decimal point. Examples of valid integer constants are:

5 −10 +25 1000 253 −26351 +36

As these examples illustrate, integers may be signed (have a leading + or − sign) or unsigned (no leading + or − sign). No commas, decimal points, or special symbols, such as the dollar sign, are allowed. Examples of invalid integer constants are:

$255.62 2,523 3. 6,243,892 1,492.89 +6.0

The largest (most positive) and smallest (most negative) integer values that can be used in a program depend on the amount of storage each computer sets aside for an integer. The more commonly used storage allocations are listed in Table 2–1. By referring to your computer's reference manual or using the `sizeof` operator introduced in Section 2.9, you can determine the actual number of bytes allocated by your computer for each integer value. For IBM 370, DEC-VAX, and PRIME computers the most positive integer allowed is 2147483647 and the most negative integer is −2147483648.[1]

Floating Point and Double Precision Constants

Both *floating point* and *double precision* constants are any signed or unsigned numbers having a decimal point. Examples of floating point and double precision numbers are:

+10.625 5. −6.2 3251.92 0.0 0.33 −6.67 +2.

Notice that the numbers 5., 0.0, and +2. are classified as floating point or double precision constants in C, while the same numbers written without a decimal point (5, 0, +2) are classified as integer constants.

The difference between floating point and double precision numbers is the amount of storage that a computer uses for each type. Most computers use twice the amount of storage for double precision numbers than for floating point numbers, which allows a double precision number to have approximately twice the

[1] It is interesting to note that in all cases the magnitude of the most negative integer allowed is always one more than the magnitude of the most positive integer. This is due to the method most commonly used to represent integers, called two's complement representation. For an explanation of two's complement representation see the Enrichment Section at the end of Chapter 1.

TABLE 2–1 Integer Values and Word Size

Word size	Maximum integer value	Minimum integer value
1 Byte	127	−128
2 Bytes	32,767	−32,768
4 Bytes	2,147,483,647	−2,147,483,648

precision of a floating point number (for this reason floating point numbers are sometimes referred to as single precision numbers). The actual storage allocation for each data type, however, depends on the particular computer. The `sizeof` operator introduced in Section 2.9 allows you to determine the amount of storage reserved by your computer for each of these data types.

Although floating point and double precision constants can be signed or unsigned, no special symbols, such as the dollar sign and the comma, are permitted. Examples of invalid floating point and double precision constants are:

5,326.25 24 123 6,459 $10.29

Exponential Notation

Floating point and double precision numbers can be written in exponential notation, which is similar to scientific notation and is useful in expressing both very large or very small numbers in compact form. The following examples illustrate how numbers with decimals can be expressed in exponential notation.

Decimal notation	Exponential notation	Scientific
1625.	1.625e3	1.625×10^3
63421.	6.3421e4	6.3421×10^4
.00731	7.31e-3	7.31×10^{-3}
.000625	6.25e-4	6.25×10^{-4}

In exponential notation the letter e stands for exponent. The number following the e represents a power of 10 and indicates the number of places the decimal point should be moved to obtain the standard decimal value. The decimal point is moved to the right if the number after the e is positive, or moved to the left if the number after the e is negative. For example, the e3 in the number 1.625e3 means move the decimal point three places to the right, so that the number becomes 1625. The e-3 in the number 7.31e-3 means move the decimal point three places to the left, so that 7.31e-3 becomes .00731.

Character Constants

The fourth basic data type recognized by C is the character constant. Characters are the letters of the alphabet, the ten digits 0 through 9, and special symbols such as + $. , − !. A single character constant is any one letter, digit, or special symbol enclosed by single quotes. Examples of valid character constants are:

'A' '$' 'b' '7' 'y' '!' 'M' 'q'

Character constants are typically stored in a computer using either the ASCII or EBCDIC codes. ASCII, pronounced AS-KEY, is an acronym for American Standard Code for Information Interchange. EBCDIC, pronounced EBB-SAH-DICK, is an acronym for Extended Binary Coded Decimal Interchange Code. These codes assign individual characters to a specific pattern of 0s and 1s. Table 2–2 lists the correspondence between bit patterns and the uppercase letters of the alphabet used by the ASCII and EBCDIC codes.

Using Table 2–2, we can determine how the character constants 'J', 'O', 'N', 'E', and 'S', for example, are stored inside a computer that uses the ASCII character code. Using the ASCII code, this sequence of characters requires five bytes of storage (one byte for each letter) and would be stored as illustrated in Figure 2–1.

TABLE 2–2 The ASCII and EBCDIC Uppercase Letter Codes

Letter	ASCII Code	EBCDIC Code	Letter	ASCII Code	EBCDIC Code
A	01000001	11000001	N	01001110	11010101
B	01000010	11000010	O	01001111	11010110
C	01000011	11000011	P	01010000	11010111
D	01000100	11000100	Q	01010001	11011000
E	01000101	11000101	R	01010010	11011001
F	01000110	11000110	S	01010011	11100010
G	01000111	11000111	T	01010100	11100011
H	01001000	11001000	U	01010101	11100100
I	01001001	11001001	V	01010110	11100101
J	01001010	11010001	W	01010111	11100110
K	01001011	11010010	X	01011000	11100111
L	01001100	11010011	Y	01011001	11101000
M	01001101	11010100	Z	01011010	11101001

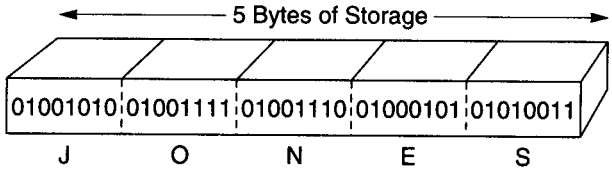


FIGURE 2–1 The Letters JONES Stored Inside a Computer

Escape Sequences

When a backslash (\) is used directly in front of a select group of characters, the backslash tells the computer to escape from the way these characters would normally be interpreted. For this reason, the combination of a backslash and these specific characters is called an escape sequence. We have already encountered an example of this in the newline escape sequence, `\n`. Table 2–3 lists other common escape sequences.

TABLE 2–3 Escape Sequences

Escape sequence	Meaning
\b	move back one space
\f	move to next page
\n	move to next line
\r	carriage return
\t	move to next tab setting
\\	backslash character
\'	single quote
\"	double quote
\nnn	treat nnn as an octal number

TABLE 2–4 The ASCII Escape Sequence Codes

C Escape sequence	Meaning	Computer code
\b	backspace	00001000
\f	form feed	00001100
\n	newline	00001010
\r	carriage return	00001101
\\	backslash	01011100
\'	single quote	00100111
\"	double quote	00100010

Although each escape sequence listed in Table 2–3 is made up of two distinct characters, the combination of the two characters with no intervening white space causes the computer to store one character code. Table 2–4 lists the ASCII code byte patterns for the escape sequences listed in Table 2–3.

Arithmetic Operations

Integers, floating point numbers, and double precision numbers may be added, subtracted, multiplied, and divided. The symbols for performing these arithmetic operations are called *arithmetic operators:*

Operation	Operator
Addition	+
Subtraction	−
Multiplication	*
Division	/
Modulus Division	%

A *simple arithmetic expression* consists of an arithmetic operator connecting two arithmetic operands in the form:

operand *operator* operand

Examples of arithmetic expressions are:

```
3 + 7
18 − 3
12.62 + 9.8
.08 * 12.2
12.6 / 2.
```

The spaces around the arithmetic operators in these examples are inserted strictly for clarity and may be omitted without affecting the value of the expression.

The value of any arithmetic expression can be displayed using the `printf( )` function. Doing this requires passing two items to `printf( )`, a control string that tells the function where and in what form the result is to be displayed, and the expression that we wish to evaluate. For example, the value of the expression 6 * 14 can be displayed using the statement

```
printf("The value of 6 times 14 is %d", 6 * 14);
```

This statement passes two arguments to the `printf( )` function. The first argument is the message `The value of 6 times 14 is %d`. The second argument is the value of the expression `6 * 14`.

The first argument passed to `printf( )` must always be a message. A message that also includes a *conversion control sequence,* such as `%d`, is termed a *control*

string.[2] Conversion control sequences have a special meaning to the `printf( )` function. They tell the function what type of value is to be displayed and where to display it. Conversion control sequences are also referred to as conversion specifications and format specifiers. A conversion control sequence always begins with a `%` symbol and ends with a conversion character (`c, d, f,` etc.). As we will see, additional formatting characters can be placed between the `%` symbol and the conversion character.

The percent sign `%` in a conversion control sequence tells `printf( )` to print a number at the place in the message where the `%` is located. The `d`, placed immediately after the `%`, tells `printf( )` that the number should be printed as an integer.

When `printf( )` sees the conversion control sequence in its control string, it substitutes the value of the next argument in place of the conversion control sequence. Since this next argument is the expression `6 * 14`, which has a value of 84, it is this value that is displayed. Also, as indicated in the example, all arguments passed to the `printf( )` function must be separated by commas. Thus, the statement

```
printf("The value of 6 times 14 is %d", 6 * 14);
```

causes the printout

```
The value of 6 times 14 is 84
```

Just as the `%d` conversion control sequence alerts `printf( )` that an integer value is to be displayed, the conversion control sequence `%f` (the `f` stands for floating point) indicates that a number with a decimal point is to be displayed. For example, the statement

```
printf("The value of 0.06 times 14.8 is %f.", 0.06 * 14.8);
```

causes the display

```
The value of 0.06 times 14.8 is 0.888000
```

As this display shows, the `%f` conversion control sequence causes `printf( )` to display six digits to the right of the decimal place. If the number does not have six decimal digits, zeros are added to the number to fill the fractional part. If the number has more than six decimal digits, the fractional part is rounded to six decimal digits.

One caution should be mentioned here: The `printf( )` function does not check the values it is given. If an integer conversion control sequence is used (`%d`, for example) and the value given the function is either a floating point or double precision number, the display will be machine dependent. Similarly, if a floating point conversion control sequence is used and the corresponding number is an integer, an unanticipated result will occur.

[2] More formally, a control string is referred to as a control specifier. We will use the more descriptive term, control string, to emphasize that a string is being referenced.

Character data is displayed using the `%c` conversion control sequence. For example, the statement

```
printf("The first letter of the alphabet is an %c.",'a');
```

causes the display

```
The first letter of the alphabet is an a.
```

Program 2-1 illustrates using `printf( )` to display the results of an expression within the statements of a complete program.

Program 2-1

```
#include <stdio.h>
main()
{
  printf("%f plus %f equals %f\n", 15.0, 2.0, 15.0 + 2.0);
  printf("%f minus %f equals %f\n",15.0, 2.0, 15.0 - 2.0);
  printf("%f times %f equals %f\n",15.0, 2.0, 15.0 * 2.0);
  printf("%f divided by %f equals %f",15.0, 2.0, 15.0 / 2.0);
}
```

The output of Program 2-1 is:

```
15.000000 plus 2.000000 equals 17.000000
15.000000 minus 2.000000 equals 13.000000
15.000000 times 2.000000 equals 30.000000
15.000000 divided by 2.000000 equals 7.500000
```

Notice that each statement in Program 2-1 passes four arguments to the `printf( )` function consisting of one control string and three values. Within each control string there are three `%f` conversion control sequences (one for each value that is to be displayed).

Expression Types

An expression that contains only integer operands is called an *integer expression*, and the result of the expression is an integer value. Similarly, an expression containing only floating point operands (single and double precision) is called a *floating point* expression, and the result of such an expression is a floating point value. An expression containing both integer and floating point operands is called a *mixed-mode* expression. Although it is better not to mix integer and floating point operands in an arithmetic operation, the resulting data type of an operation is determined by the following rules:

1. If both operands are integers, the result of the operation is an integer.
2. If one operand is a floating point or double precision value, the result of the operation is a double precision number.

Notice that the result of an arithmetic expression is never a floating point number because the computer temporarily converts all floating point numbers to double precision numbers when arithmetic is being done.

Integer Division

The division of two integers can produce rather strange results for the unwary. For example, dividing the integer 15 by the integer 2 yields an integer result. Since integers cannot contain a fractional part, the expected result, 7.5, is not obtained. In C, the fractional part of the result obtained when dividing two integers is dropped (truncated). Thus, the value of 15/2 is 7, the value of 9/4 is 2, and the value of 19/5 is 3.

There are times when we would like to retain the remainder of an integer division. To do this C provides an arithmetic operator that captures the remainder when two integers are divided. This operator, called the modulus operator, has the symbol %. The modulus operator can be used only with integers. For example,

9 % 4 is 1
17 % 3 is 2
14 % 2 is 0

A Unary Operator (Negation)

Besides the binary operators for addition, subtraction, multiplication, and division, C also provides unary operators. One of these unary operators uses the same symbol that is used for binary subtraction (−). The minus sign used in front of a single numerical operand negates (reverses the sign of) the number.

Table 2–5 summarizes the six arithmetic operations we have described so far and lists the data type of the result produced by each operator based on the data type of the operands involved.

Operator Precedence and Associativity

Besides such simple expressions as 5 + 12 and .08 * 26.2, we frequently need to create more complex arithmetic expressions. C, like most other programming languages, requires that certain rules be followed when writing expressions containing more than one arithmetic operator. These rules are:

1. Two binary arithmetic operator symbols must never be placed side by side.
   For example, 5 * %6 is invalid because the two operators * and % are placed next to each other.
2. Parentheses may be used to form groupings, and all expressions enclosed within parentheses are evaluated first.
   For example, in the expression (6 + 4) / (2 + 3), the 6 + 4 and 2 + 3 are evaluated first to yield 10 / 5. The 10 / 5 is then evaluated to yield 2.
   Sets of parentheses may also be enclosed by other parentheses. For example, the expression (2 * (3 + 7)) / 5 is valid. When parentheses are used within

TABLE 2–5 Summary of Arithmetic Operators

Operation	Operator	Type	Operand	Result
Addition	+	Binary	Both integers	Integer
			One operand not an integer	Double precision
Subtraction	–	Binary	Both integers	Integer
			One operand not an integer	Double precision
Multiplication	*	Binary	Both integers	Integer
			One operand not an integer	Double precision
Division	/	Binary	Both integers	Integer
			One operand not an integer	Double precision
Modulus	%	Binary	Both integers	Integer
Negation	–	Unary	One integer	Integer
			One floating point or double precision operand	Double precision

parentheses, the expressions in the innermost parentheses are always evaluated first. The evaluation continues from innermost to outermost parentheses until the expressions of all parentheses have been evaluated. The number of right-facing parentheses, (, must always equal the number of left-facing parentheses,), so that there are no unpaired sets.

3. Parentheses cannot be used to indicate multiplication. The multiplication operator, *, must be used.

   For example, the expression (3 + 4) (5 + 1) is invalid. The correct expression is (3 + 4) * (5 + 1).

As a general rule, parentheses should be used to specify logical groupings of operands and to indicate clearly to both the computer and programmers the intended order of arithmetic operations. In the absence of parentheses, expressions containing multiple operators are evaluated by the priority, or precedence, of each operator. Table 2–6 lists both the precedence and associativity of the operators considered in this section.

TABLE 2–6 Operator Precedence and Associativity

Operator	Associativity
unary –	right to left
* / %	left to right
+ –	left to right

The precedence of an operator establishes its priority relative to all other operators. Operators at the top of Table 2–6 have a higher priority than operators at the bottom of the table. In expressions with multiple operators, the operator with the higher precedence is used before an operator with a lower precedence. For example, in the expression 6 + 4 / 2 + 3, the division is done before the addition, yielding an intermediate result of 6 + 2 + 3. The additions are then performed to yield a final result of 11.

Expressions containing operators with the same precedence are evaluated according to their associativity. This means that evaluation is either from left to right or from right to left as each operator is encountered. For example, in the expression 8 + 5 * 7 % 2 * 4, the multiplication and modulus operator are of higher precedence than the addition operator and are evaluated first. Both of these operators, however, are of equal precedence. Therefore, these operators are evaluated according to their left-to-right associativity, yielding

$$\begin{aligned} 8 + 5 * 7 \,\%\, 2 * 4 &= \\ 8 + 35 \,\%\, 2 * 4 &= \\ 8 + 1 * 4 &= \\ 8 + 4 &= 12 \end{aligned}$$

Exercises 2.1

1. Determine data types appropriate for the following data:
   a. the average of four grades
   b. the number of days in a month
   c. the length of the Golden Gate Bridge
   d. the numbers in a state lottery
   e. the distance from Brooklyn, N.Y. to Newark, N.J.
   f. the names in a mailing list
2. Convert the following numbers into standard decimal form:

   6.34e5 1.95162e2 8.395e1 2.95e-3 4.623e-4

3. Write the following decimal numbers using exponential notation:

   126. 656.23 3426.95 4893.2 .321 .0123 .006789

4. Listed below are correct algebraic expressions and incorrect C expressions corresponding to them. Find the errors and write corrected C expressions.

Algebra	C expression
a. (2)(3) + (4)(5)	`(2)(3) + (4)(5)`
b. $\frac{6+18}{2}$	`6 + 18 / 2`
c. $\frac{4.5}{12.2 - 3.1}$	`4.5 / 12.2 - 3.1`

d. 4.6(3.0 + 14.9)	`4.6(3.0 + 14.9)`
e. (12.1 + 18.9)(15.3 - 3.8)	`(12.1 + 18.9)(15.3 - 3.8)`

5. Determine the value of the following integer expressions:

a. 3 + 4 * 6
b. 3 * 4 / 6 + 6
c. 2 * 3 / 12 * 8 / 4
d. 10 * (1 + 7 * 3)
e. 20 − 2 / 6 + 3
f. 20 − 2 / (6 + 3)
g. (20 − 2) / 6 + 3
h. (20 − 2) / (6 + 3)
i. 50 % 20
j. (10 + 3) % 4

6. Determine the value of the following floating point expressions:

a. 3.0 + 4.0 * 6.0
b. 3.0 * 4.0 / 6.0 + 6.0
c. 2.0 * 3.0 / 12.0 * 8.0 / 4.0
d. 10.0 * (1.0 + 7.0 * 3.0)
e. 20.0 − 2.0 / 6.0 + 3.0
f. 20.0 − 2.0 / (6.0 + 3.0)
g. (20.0 − 2.0) / 6.0 + 3.0
h. (20.0 − 2.0) / (6.0 + 3.0)

7. Evaluate the following mixed-mode expressions and list the data type of the result. In evaluating the expressions be aware of the data types of all intermediate calculations.

a. 10.0 + 15 / 2 + 4.3
b. 10.0 + 15.0 / 2 + 4.3
c. 3.0 * 4 / 6 + 6
d. 3 * 4.0 / 6 + 6
e. 20.0 − 2 / 6 + 3
f. 10 + 17 * 3 + 4
g. 10 + 17 / 3. + 4
h. 3.0 * 4 % 6 + 6
i. 10 + 17 % 3 + 4.

8. Assume that amount has the integer value 1, *m* has the integer value 50, *n* has the integer value 10, and *p* has the integer value 5. Evaluate the following expressions:

a. *n* / *p* + 3
b. *m* / *p* + *n* − 10 * amount
c. *m* − 3 * *n* + 4 * amount
d. amount / 5
e. 18 / *p*
f. −*p* * *n*
g. −*m* / 20
h. (*m* + *n*) / (*p* + amount)
i. *m* + *n* / *p* + amount

9. Repeat Exercise 8 assuming that amount has the real value 1.0, *m* has the real value 50.0, *n* has the real value 10.0, and *p* has the real value 5.0.

10. Using the system reference manuals for your computer, determine the character code used by your computer.

11. Determine the output of the following program:

```
#include <stdio.h>
main()  /* a program illustrating integer truncation */
{
  printf("answer1 is the integer %d", 9/4);
  printf("\nanswer2 is the integer %d", 17/3);
}
```

12. Determine the output of the following program:

```
#include <stdio.h>
main()  /* a program illustrating the % operator */
{
  printf("The remainder of 9 divided by 4 is %d", 9 % 4);
  printf("The remainder of 17 divided by 3 is %d", 17 % 3);
}
```

13. Write a C program that displays the results of the expressions 3.0 * 5.0, 7.1 * 8.3 − 2.2, and 3.2 / (6.1 * 5). Calculate the value of these expressions manually to verify that the displayed values are correct.

14. Write a C program that displays the results of the expressions 15 / 4, 15 % 4, and 5 * 3 − (6 * 4). Calculate the value of these expressions manually to verify that the display produced by your program is correct.

15a. Show how the name KINGSLEY would be stored inside a computer that uses the ASCII code. That is, draw a figure similar to Figure 2–1 for the letters KINGSLEY.
b. Show how the name KINGSLEY would be stored inside a computer that uses the EBCDIC code.

16a. Repeat Exercise 15a using the letters of your own last name.
b. Repeat Exercise 15b using the letters of your own last name.

Expanding Your Skills

17. Enter, compile, and run Program 2-1 on your computer system.

18. Since computers use different representations for storing integer, floating point, double precision, and character values, discuss how a program might alert the computer to the data types of the various values it will be using.

Note: ***For the following exercise you should have an understanding of basic computer storage concepts. Specifically, if you are unfamiliar with the concept of a byte, refer to the enrichment section at the end of Chapter 1 before doing the next exercise.***

19. Although the total number of bytes varies from computer to computer, memory sizes of 65,536 to more than 1 million bytes are not uncommon. In computer language, the letter K is used to represent the number 1024, which is 2 raised to the 10th power. Thus, a memory size of 64K is really 64 times 1024, or 65,536 bytes, and a memory size of 512K consists of 512 times 1024, or 524,288 bytes. Using this information, calculate the actual number of bytes in:

a. a memory containing 64K bytes
b. a memory containing 128K bytes
c. a memory containing 192K bytes
d. a memory containing 256K bytes
e. a memory consisting of 64K words, where each word consists of 2 bytes
f. a memory consisting of 64K words, where each word consists of 4 bytes
g. a floppy diskette that can store 360K bytes

20. Although we have concentrated on only integer, floating point, and double precision numbers, C allows characters and integers to be added or subtracted. This can be done because C always converts a character to an equivalent integer value whenever a character is used in an arithmetic expression. Thus, characters and integers can be freely mixed in such expressions. For example, if your computer uses the ASCII code, the expression 'a' + 1 equals 'b', and 'z' − 1 equals 'y'. Similarly, 'A' + 1 is 'B', and 'Z' − 1 is 'Y'. With this

as background, determine the character results of the following expressions (assume that all characters are stored using the ASCII code).

a. 'm' − 5
b. 'm' + 5
c. 'G' + 6
d. 'G' − 6
e. 'b' − 'a'
f. 'g' − 'a' + 1
g. 'G' − 'A' + 1

21a. The table in Appendix F lists the integer values corresponding to each letter stored using the ASCII code. Using this table, notice that the uppercase letters consist of contiguous codes starting with an integer value of 65 for A and ending with 90 for the letter Z. Similarly, the lowercase letters begin with the integer value of 97 for the letter a and end with 122 for the letter z. With this as background, determine the character value of the expressions 'A' + 32 and 'Z' + 32.
b. Using Appendix F, determine the integer value of the expression 'à' − 'A'.
c. Using the results of Exercises 21a and 21b, determine the character value of the following expression, where uppercase letter can be any uppercase letter from A to Z:

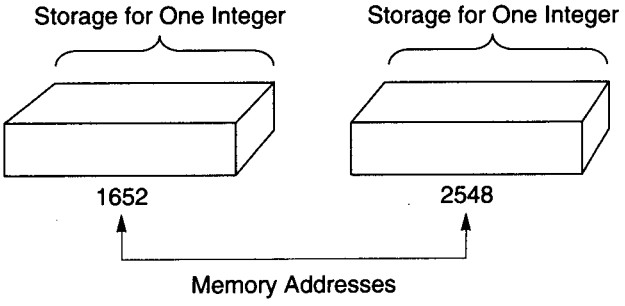
uppercase letter + 'a' − 'A'

2.2 Variables and Declaration Statements

All integers, numbers, floating point numbers, and values used in a computer program are stored and retrieved from the computer's memory unit. Conceptually, individual memory locations in the memory unit are arranged like the rooms in a large hotel. Like hotel rooms, each memory location has a unique address ("room number"). Before high-level languages such as C existed, memory locations were referenced by their addresses. For example, to store the integer values 45 and 12 in the memory locations 1652 and 2548 (see Figure 2–2), respectively, required instructions equivalent to

put a 45 in location 1652
put a 12 in location 2548

FIGURE 2–2 Enough Storage for Two Integers



To add the two numbers just stored and save the result in another memory location, for example at location 3000, required a statement comparable to

add the contents of location 1652
to the contents of location 2548
and store the result location 3000

Clearly this method of storage and retrieval is a cumbersome process. In high-level languages like C, symbolic names are used in place of actual memory addresses. Symbolic names used in this manner are called *variables*. A variable is simply a name given by the programmer to a memory storage location. The term variable is used because the value stored in the variable can change, or vary. For each name that the programmer uses, the computer keeps track of the actual memory address corresponding to that name. In our hotel room analogy, this is equivalent to putting a name on the door of a room and referring to the room by this name, such as the Blue Room, rather than using the actual room number.

In C the selection of variable names is left to the programmer, as long as the variable name is chosen according to the same rules used for selecting function names given in the previous chapter. Thus, a variable name can consist of no more than 31 letters, digits, or underscore, the first of which must be a letter or underscore and cannot be a keyword (see Table 1–2).

As with function names, variable names should be mnemonics that give some indication of the variable's use. For example, a good name for a variable used to store a value that is the total of some other values would be `sum` or `total`. Similarly the variable name `width` is a good choice if the value stored in the variable represents a width. Variable names that give no indication of the value stored, such as `r2d2`, `linda`, `bill`, and `getum` should not be selected.

Now assume the first memory location illustrated in Figure 2–2, that has address 1652, is given the name `num1`. Also assume that memory location 2548 is given the variable name `num2`, and memory location 3000 is given the variable name `total`, as illustrated in Figure 2–3.

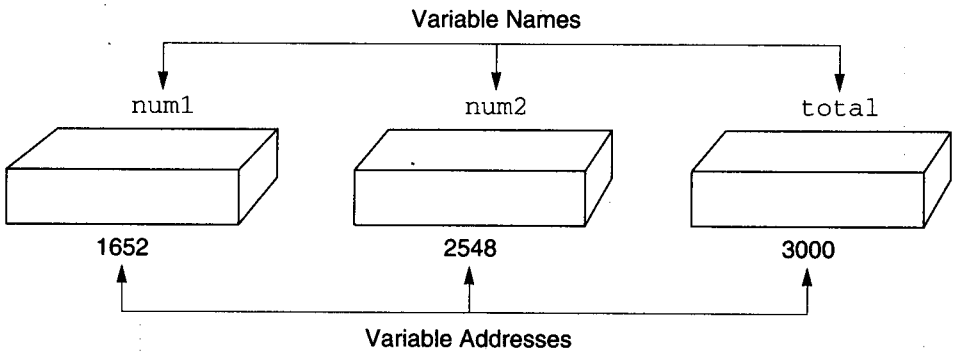
Using these variable names, the operations of storing 45 in location 1652, storing 12 in location 2548, and adding the contents of these two locations is accomplished by the C statements

```
num1 = 45;
num2 = 12;
total = num1 + num2;
```

Each of these three statements is called an assignment statement because it tells the computer to assign (store) a value into a variable. Assignment statements always have an equal (=) sign and one variable name immediately to the left of this sign. The value on the right of the equal sign is determined first and this value is assigned to the variable on the left of the equal sign. The blank spaces in the assignment statements are inserted for readability. We will have much more to say about assignment statements in the next section, but for now we can use them to store values in variables.

A variable name is useful because it frees the programmer from concern over where data are physically stored inside the computer. We simply use the variable name and let the computer worry about where in memory the datum is actually stored. Before storing a value into a variable, however, we must clearly define

FIGURE 2–3 Naming Storage Locations



the type of data that is to be stored in it. This requires telling the compiler, in advance, the names of the variables that will be used for integers, the names that will be used for floating point numbers, and the names that will be used to store the other C data types.

Declaration Statements

Naming a variable and specifying the data type that can be stored in it are accomplished using declaration statements. A declaration statement has the general form:

data-type variable name;

where data-type designates a valid C data type and variable name is a user-selected variable name. For example, variables used to hold integer values are declared using the reserved word `int` to specify the data type and have the form:

```
int variable name;
```

Thus, the declaration statement

```
int total;
```

declares `total` as the name of a variable capable of storing an integer value. [3]

Variables used to hold floating point values are declared using the keyword `float`, while variables that will be used to hold double precision values are declared using the keyword `double`. For example, the statement

```
float firstnum;
```

[3] In addition to the keyword `int` used to specify an integer, the keyword `long` specifies a long integer, which typically doubles the size of the allowable integer value that can be assigned to the declared variable. Also, the keywords `unsigned int` are used to specify an integer that can only store nonnegative numbers. Long integers and unsigned integers are displayed using the `%ld` and `%u` conversion control sequences, respectively, in the `printf( )` function.

declares `firstnum` as a variable that can be used to store a floating point number. Similarly, the statement

```
double secnum;
```

declares that the variable `secum` will be used to store a double precision number.

Declaration statements within a function appear immediately after the opening brace of a function, and like all C statements must end with a semicolon. A `main( )` function containing declaration statements has the general form:

```
main ()
{
  declaration statements;

  other statements;
}
```

Program 2-2 illustrates the declaration and use of four floating point variables. The `printf()` function is then used to display the contents of one of these variables.

Program 2-2

```
#include <stdio.h>
main()
{
  float grade1;  /* declare grade1 as a float variable */
  float grade2;  /* declare grade2 as a float variable */
  float total;   /* declare total as a float variable */
  float average; /* declare average as a float variable */

  grade1 = 85.5;
  grade2 = 97.0;
  total = grade1 + grade2;
  average = total/2.0;  /* divide the total by 2.0 */
  printf("The average grade is %f\n",average);
}
```

The placement of the declaration statements in Program 2-2 is straightforward, although we will shortly see that individual declaration statements for the same data type are typically combined into a single declaration statement. When Program 2-2 is run, the following output is displayed:

```
The average grade is 91.250000
```

Two comments with respect to the `printf()` function call made in Program 2-2 should be mentioned here. If a variable name is one of the arguments passed to a function, as it is to `printf()` in Program 2-2, the function only receives a copy of the value stored in the variable. It does not receive the variable's name. When the program sees a variable name in the function parentheses, it first goes to the variable and retrieves the value stored. It is this value that is passed to the function. Thus, when a variable is included in the `printf()` argument list, `printf()` receives the value stored in the variable and then displays this value. Internally, `printf()` has no knowledge of where the value it receives came from or the variable name under which the value was stored.

Although this procedure for passing data into a function may seem surprising, it is really a safety procedure for ensuring that a called function does not have access to the original variable. This guarantees that the called function cannot inadvertently change data in a variable declared outside itself. We will have more to say about this in Chapter 7 when we examine and begin writing our own functions.

The second comment concerns the use of the `%f` conversion control sequence in Program 2-2. Although this conversion control sequence works for both floating point and double precision numbers, the conversion control sequence `%lf` may also be used for displaying the values of double precision variables. The letter `l` indicates that the number is a long floating point number, which is what a double precision number really is. Omitting the `l` conversion character has no effect on the `printf()` function when double precision values are displayed. As we shall see, however, it is essential in entering double precision values when the input function `scanf()`, introduced in the next chapter, is used.

Just as integers, floating point, and double precision variables must be declared before they can be used, a variable used to store a character must also be declared. Character variables are declared using the reserved word `char`. For example, the declaration

```
char ch;
```

declares `ch` to be a character variable.

Multiple Declarations

Variables having the same data type can always be grouped together and declared using a single declaration statement. The common form of such a declaration is:

data-type variable list;

For example, the four separate declarations

```
float grade1;
float grade2;
float total;
float average;
```

can be replaced by the single declaration statement

```
float grade1, grade2, total, average;
```

Similarly, the two integer declarations

```
int num1;
int num2;
```

can be replaced with the single declaration statement

```
int num1, num2;
```

Notice that declaring multiple variables in a single declaration requires that the data type of the variables be given only once, that all the variables be separated by commas, and that only one semicolon be used to terminate the declaration. The space after each comma is inserted for readability and is not required.

Declaration statements can also be used to store an initial value into each declared variable. For example, the declaration statement

```
int num1 = 15;
```

both declares the variable `num1` as an integer variable and sets the value of `15` into the variable. The first time a value is stored in a variable the variable is said to be *initialized.* Thus, in this example it is correct to say that the variable `num1` has been initialized to `15`. Similarly, the declaration statement

```
float grade1 = 87.0, grade2 = 93.5, total;
```

declares three floating point variables and initializes two of them. Constants, expressions using only constants (such as 87.0 + 12.2), and expressions using constants and previously initialized variables can all be used as initializers within a declaration statement. For example, Program 2-2 with initialization of the variables within their declaration statements would appear as:

```
#include <stdio.h>
main()
{
  float grade1 = 85.5, grade2 = 97.0, total, average;

  total = grade1 + grade2;
  average = total/2.0;  /* divide the total by 2.0 */
  printf("The average grade is %f\n",average);
}
```

Notice the blank space after the declaration statement. Placing a blank line after variable declarations is a common programming practice that improves both a program's appearance and its readability. We will adopt this practice for all of our programs.

Specifying Storage Allocation

Declaration statements perform both software and hardware tasks. From a software perspective, declaration statements always provide a convenient, up-front list of all variables and their data types. In this software role, variable declarations also eliminate an otherwise common and troublesome error caused by the misspelling of a variable's name within a program. For example, assume that a variable named `distnce` is declared and initialized using the statement

```
int distance = 26;
```

Now assume that this variable is inadvertently misspelled in the statement

```
mpg = distce / gallons;
```

In languages that do not require variable declarations, the program would treat `distnce` as a new variable and either assign an initial value of zero to the variable or use whatever value happened to be in the variable's storage area. In either case a value would be calculated and assigned to `mpg`, and finding the error or even knowing that an error occurred could be extremely troublesome. Such errors are impossible in C, because the compiler will flag `distnce` as an undeclared variable. The compiler cannot, of course, detect when one declared variable is typed in place of another declared variable.

In addition to their software role, declaration statements can also perform a distinct hardware task. Since each data type has its own storage requirements, the computer can only allocate sufficient storage for a variable after it knows the variable's data type. Because variable declarations provide this information, they can be used to force the computer to reserve sufficient physical memory storage for each variable. Declaration statements used for this hardware purpose are also called *definition statements*, because they define or tell the computer how much memory is needed for data storage.

All the declaration statements we have encountered so far have also been definition statements. Later, we will see cases of declaration statements that do not cause any new storage to be allocated and are used simply to declare or alert the program to the data types of previously created and existing variables.

Figure 2–4 illustrates the series of operations set in motion by declaration statements that also perform a definition role. The figure shows that definition statements (or, if you prefer, declaration statements that also cause memory to be allocated) "tag" the first byte of each set of reserved bytes with a name. This name is, of course, the variable's name and is used by the computer to correctly locate the starting point of each variable's reserved memory area.

Within a program, after a variable has been declared, it is typically used by a programmer to refer to the contents of the variable (that is, the variable's value). Where in memory this value is stored is generally of little concern to the programmer. The computer, however, must be concerned with where each value is stored and with correctly locating each variable. In this task the computer uses the variable name to locate the first byte of storage previously allocated to the variable. Knowing the variable's data type then allows the computer to store or retrieve the correct number of bytes.

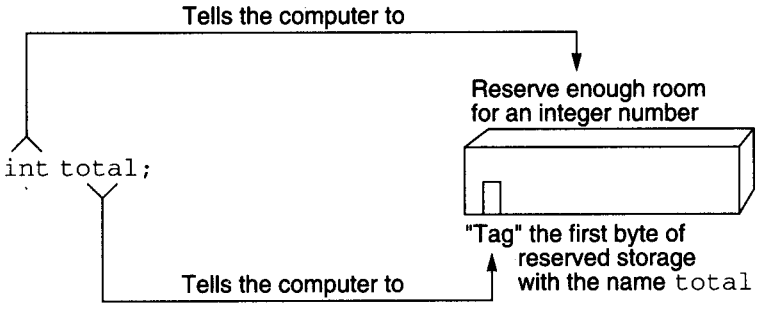


FIGURE 2–4a Defining the Integer Variable Named `total`

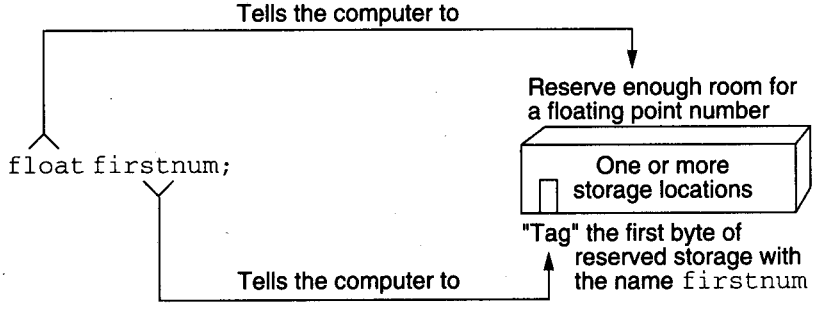


FIGURE 2–4b Defining the Floating Point Variable Named `firstnum`

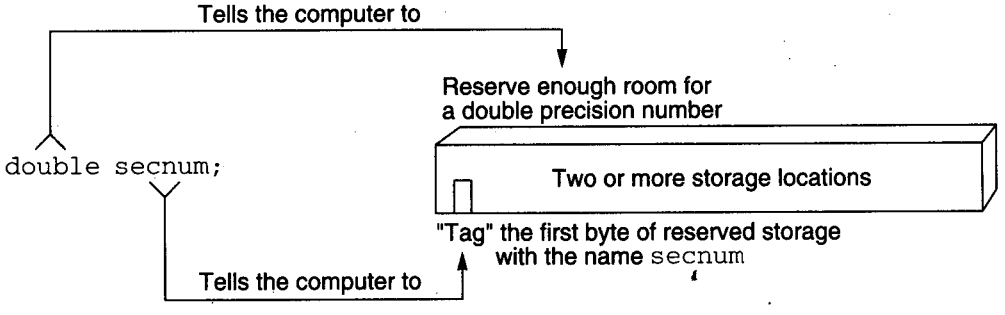


FIGURE 2–4c Defining the Double Precision Variable Named `secnum`

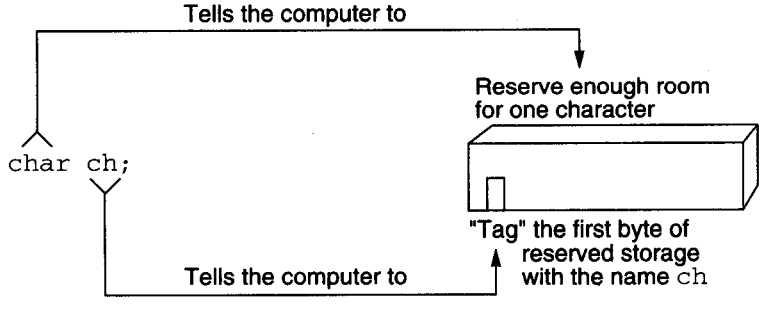


FIGURE 2–4d Defining the Character Variable Named `ch`

Exercises 2.2

1. State whether the following variable names are valid or not. If they are invalid, state the reason why.

prod_a	c1234	abcd	_c3	12345
newbal	while	$total	new bal	a1b2c3d4
9ab6	sum.of	average	grade1	fin_grad

2. State whether the following variable names are valid or not. If they are invalid, state the reason why. Also indicate which of the valid variable names should not be used because they convey no information about the variable.

salestax	a243	r2d2	first_num	cc_a1
harry	sue	c3p0	average	sum
maximum	okay	a	awesome	goforit
3sum	for	tot.a1	c$five	netpay

3a. Write a declaration statement to declare that the variable `count` will be used to store an integer.

b. Write a declaration statement to declare that the variable `grade` will be used to store a floating point number.

c. Write a declaration statement to declare that the variable `yield` will be used to store a double precision number.

d. Write a declaration statement to declare that the variable `initial` will be used to store a character.

4. Write declaration statements for the following variables.

a. `num1`, `num2`, and `num3` used to store integer numbers

b. `grade1`, `grade2`, `grade3`, and `grade4` used to store floating point numbers

c. `tempa`, `tempb`, and `tempc` used to store double precision numbers

d. `ch`, `let1`, `let2`, `let3`, and `let4` used to store character types

5. Write declaration statements for the following variables.

a. `firstnum` and `secnum` used to store integers

b. `price`, `yield`, and `coupon` used to store floating point numbers

c. `maturity` used to store a double precision number

6. Rewrite each of these declaration statements as three individual declarations.

a. `int month, day = 30, year;`

b. `double hours, rate, otime = 15.62;`

c. `float price, amount, taxes;`

d. `char in_key, ch, choice = 'f';`

7a. Determine what each statement causes to happen in the following program.

```
#include <stdio.h>
main()
{
  int num1;
  int num2;
  int total;

  num1 = 25;
  num2 = 30;
  total = num1 + num2;
  printf("The total of %d and %d is %d\n.",num1,num2,total);
}
```

b. What is the output that will be printed when the program listed in Exercise 7a is run?

8. Write a C program that stores the sum of the integer numbers 12 and 33 in a variable named sum. Have your program display the value stored in sum.

9. Write a C program that stores the value 16 in the integer variable `length` and the value 18 in the integer variable `width`. Have your program calculate the value assigned to the variable perimeter, using the assignment statement

```
perimeter = 2*(length + width);
```

and print out the value stored in the variable `perimeter`. Make sure to declare all the variables as integers at the beginning of the `main()` function.

10. Write a C program that stores the integer value 16 in the variable num1 and the integer value 18 in the variable num2. (Make sure to declare the variables as integers.) Have your program calculate the total of these numbers and their average. The total should be stored in an integer variable named `total` and the `average` in an integer variable named average. (Use the statement `average = total/2.0;` to calculate the average.) Use the `printf()` function to display the total and average.

11. Repeat Exercise 10, but store the number 15 in num1 instead of 16. With a pencil, write down the average of num1 and num2. What do you think your program will store in the integer variable that you used for the average of these two numbers? How can you ensure that the correct answer will be printed for the average?

12. Write a program that stores the number 105.62 in the variable `firstnum`, 89.352 in the variable secnum, and 98.67 in the variable `thirdnum`. (Make sure to declare the variables first as either float or double.) Have your program calculate the total of the three numbers and their average. The total should be stored in the variable `total` and the average in the variable `average`. (Use the statement `average = total /3.0;` to calculate the average.) Use the `printf()` function to display the total and average.

13. Every variable has at least two items associated with it. What are these two items?

Note for Exercises 14 through 16: ***Assume that a character requires one byte of storage, an integer two bytes, a floating point number four bytes, a double precision number eight bytes, and that variables are assigned storage in the order they are declared.***

14a. Using Figure 2–5 and assuming that the variable name rate is assigned to the byte having memory address 159, determine the addresses corresponding to each variable declared in the following statements. Also fill in the appropriate bytes with the initialization data included in the declaration statements (use letters for the characters, not the computer codes that would actually be stored).

```
float rate;
char ch1 = 'w', ch2 = 'o', ch3 = 'w', ch4 = '!';
double taxes;
int num, count = 0;
```

b. Repeat Exercise 14a, but substitute the actual byte patterns that a computer using the ASCII code would use to store the characters in the variables `ch1`, `ch2`, `ch3`, and `ch4`. (*Hint:* Use Table 2–2.)

15a. Using Figure 2–5 and assuming that the variable named cn1 is assigned to the byte at memory address 159, determine the addresses corresponding to each variable declared in the following statements. Also fill in the appropriate bytes with the initialization data included in the declaration statements (use letters for the characters and not the computer codes that would actually be stored).

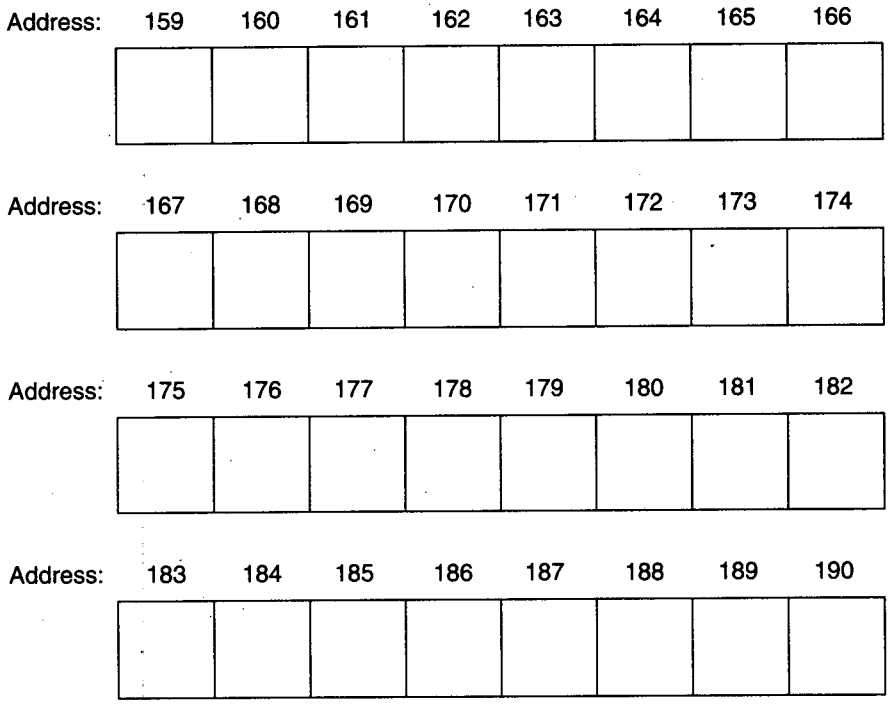


FIGURE 2–5 Memory Bytes for Exercises 14, 15, and 16

```
char cn1 = 'a', cn2 = ' ', cn3 = 'b', cn4 = 'u', cn5 = 'n';
char cn6 = 'c', cn7 = 'h', key = '\\', sch = '\'', inc = 'o';
char inc1 = 'f';
```

b. Repeat Exercise 15a, but substitute the actual byte patterns that a computer using the ASCII code would use to store the characters in each of the declared variables. (*Hint:* Use Table 2–2.)

16. Using Figure 2–5 and assuming that the variable name `miles` is assigned to the byte at memory address 159, determine the addresses corresponding to each variable declared in the following statements:

```
float miles;
int count, num;
double dist, temp;
```

2.3 Assignment Statements

The most basic C statement for both assigning values to variables and performing computations is the assignment statement. This statement has the general form

variable = operand;

The simplest operand in C is a single constant. In each of the following assignment statements, the expression to the right of the equal sign is a constant:

```
length = 25;
width = 17.5;
```

In each of these assignment statements the value of the constant to the right of the equal sign is assigned to the variable on the left side of the equal sign. It is extremely important to note that the equal sign in C does not have the same meaning as an equal sign in algebra. The equal sign in an assignment statement tells the computer first to determine the value of the operand to the right of the equal sign and then to store (or assign) that value in the variable to the left of the equal sign. In this regard, the C statement `length = 25;` is read "length is assigned the value 25." The blank spaces in the assignment statement are inserted for readability only.

Recall from the previous section that an initial value can be assigned to a variable when it is declared. If an initialization is not done from within the declaration statement, the variable is initialized the first time a value is assigned using an assignment statement. Subsequent assignment statements can, of course, be used to change the value assigned to a variable. For example, assume the following statements are executed one after another and that no value has been assigned to `slope` previously:

```
slope = 3.7;
slope = 6.28;
```

The first assignment statement assigns the value of `3.7` to the variable named `slope`. Since this is the first time a value is assigned to this variable it is also correct to say that "slope is initialized to 3.7." The next assignment statement causes the computer to assign a value of `6.28` to `slope`. The `3.7` that was in slope is simply erased and replaced with the new value of `6.2`, because a variable can store only one value at a time. It is sometimes useful to think of the variable to the left of the equal sign as a temporary parking spot in a huge parking lot. Just as an individual parking spot can be used only by one car at a time, each variable can store only one value at a time. The "parking" of a new value in a variable automatically causes the computer to remove any value previously parked there.

In addition to being a constant, the operand to the right of the equal sign in an assignment statement can be a variable or any valid C expression. An *expression* is any combination of constants and variables that can be evaluated to yield a result. Thus, the expression in an assignment statement can be used to perform calculations using the arithmetic operators introduced in Section 2.1. Examples of assignment statements using expressions containing these operators are:

```
sum = 3 + 7;
diff = 15 - 6;
product = 0.05 * 14.6;
tally = count + 1;
newtotal = 18.3 + total;
```

```
taxes = 0.06 * amount
totwet = weight * factor;
average = sum /items;
slope = (y2 - y1) / (x2 - x1);
```

As always in an assignment statement, the computer first calculates the value of the expression to the right of the equal sign and then stores this value in the variable to the left of the equal sign. For example, in the assignment statement `totwet = weight * factor` the expression `weight * factor` is first evaluated to yield a result. This result, which is a number, is then stored in the variable `totwet`.

In writing assignment statements, you must be aware of two important considerations. Since the expression to the right of the equal sign is evaluated first, all variables used in the expression must be initialized if the result is to make sense. For example, the assignment statement `totwet = weight * factor` only causes a valid number to be stored in `totwet` if the programmer first takes care to put valid numbers in `weight` and `factor`.

Thus, the sequence of statements

```
weight = 155.0;
factor = 1.06;
totwet = weight * factor;
```

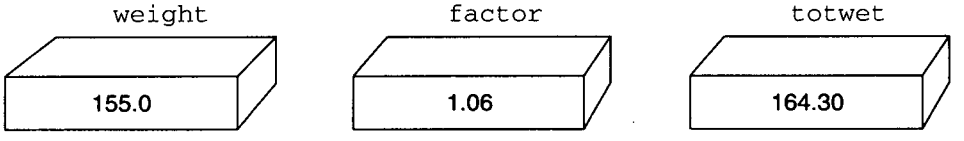
ensures that we know the values being used to obtain the result that will be stored in the variable to the left of the equal sign. Figure 2–6 illustrates the values stored in the variables `weight`, `factor`, and `totwet`.

The second consideration to keep in mind is that since the value of an expression is stored in the variable to the left of the equal sign, there must be only one variable listed in this position. For example, the assignment statement

```
amount + 1769 = 1462 + 10 - 24;
```

is invalid. The right–hand expression evaluates to the integer `1448`, which can only be stored in a variable. Since `amount + 1769` is not the valid name of a memory location (it is not a valid variable name), the computer does not know where to store the value `1448`. Program 2-3 illustrates the use of assignment statements to calculate the volume of a cylinder. As illustrated in Figure 2–7, the volume of a cylinder is determined by the formula volume = $\pi r^2 h$, where r is the radius of the cylinder, h is the height, and π is the constant 3.1416 (accurate to four decimal places).

FIGURE 2–6 Values Stored in the Variables



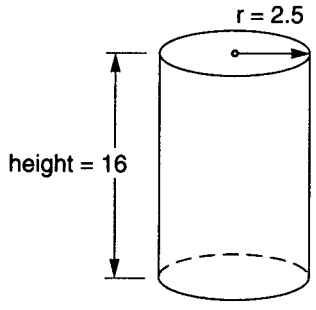


FIGURE 2–7 Determining the Volume of a Cylinder

Program 2-3

```
#include <stdio.h>
main( )
{
/* this program calculates the volume of a cylinder, */
/* given its radius and height                       */
  float radius, height, volume;

  radius = 2.5;
  height = 16.0;
  volume = 3.1416 * radius * radius * height;
  printf("/nthe volume of the cylinder is %f", volume);
}
```

When Program 2-3 is compiled and executed, the output is:

```
the volume of the cylinder is 314.160000
```

Notice the order in which statements are executed in Program 2-3. The program begins with the keyword `main` and continues sequentially, statement by statement, until the closing brace. All computer programs execute in this manner. The computer works on one statement at a time, executing that statement with no knowledge of what the next statement will be. This explains why all variables used in an expression must have values assigned to them before the expression is evaluated. When the computer executes the statement

```
volume = 3.1416 * radius * radius * height;
```

in Program 2-3, it uses whatever value is stored in the variables `radius` and `height` at the time the assignment statement is executed.[4] If no values have been

[4] Since C does not have an exponentiation operator, the square of the radius is obtained by the term radius * radius. In Section 3.1 we introduce C's power function `pow( )`, which allows us to raise a number to a power.

specifically assigned to these variables before they are used in the assignment statement, the computer uses whatever values happen to occupy these variables when they are referenced (on some systems all variables are automatically initialized to zero). The computer does not "look ahead" to see that you might assign values to these variables later in the program.

It is important to realize that in C, the equal sign, =, used in assignment statements is itself an operator, *which differs from the way most other high–level languages process this symbol.* In C, the = symbol is called the *assignment operator.* Since the equal sign is an operator in C, multiple assignments are possible in the same statement. For example, in the statement `a = b = c = 25;` all the assignment operators have the same precedence. Since the assignment operator has a right–to–left associativity, the final evaluation proceeds in the sequence

```
c = 25;
b = c;
a = b;
```

This has the effect of assigning the number `25` to each of the variables individually, which can be represented as

```
a = (b = (c = 25));
```

Thus, the single statement `a = b = c = 25;` is equivalent to the three individual statements

```
c = 25;
b = 25;
a = 25;
```

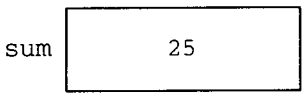
Assignment Variations

Although only one variable is allowed immediately to the left of an equal sign in an assignment expression, the variable on the left of the equal sign can also be used on the right of the equal sign. For example, the assignment expression `sum = sum + 10` is valid. Clearly, as an algebra equation sum could never be equal to itself plus 10. But in C, the expression `sum = sum + 10` is not an equation—it is an expression that is evaluated in two major steps. The first step is to calculate the value of `sum + 10`. The second step is to store the computed value in `sum`. See if you can determine the output of Program 2-4.

The assignment statement `sum = 25;` tells the computer to store the number `25` in `sum`, as shown in Figure 2–8.

The first call to `printf()` causes the value stored in sum to be displayed by

FIGURE 2–8 The Integer `25` is Stored in `sum`



Program 2-4

```
#include <stdio.h>
main( )
{
  int sum;

  sum = 25;
  printf("\nThe number stored in sum is %d.",sum);
  sum = sum + 10;
  printf("\nThe number now stored in sum is %d.",sum);
}
```

the message `The number stored in sum is 25.` The second assignment statement in Program 2-4, `sum = sum + 10;` causes the computer to retrieve the `25` stored in `sum` and add `10` to this number, yielding the number `35`. The number `35` is then stored in the variable on the left side of the equal sign, which is the variable `sum`. The `25` that was in `sum` is simply overwritten with the new value of `35`, as shown in Figure 2–9.

Assignment expressions like `sum = sum + 25`, which use the same variable on both sides of the assignment operator, can be written using the following assignment operators:

```
+=    -=    *=    /=    %=
```

For example, the expression `sum = sum + 10` can be written as `sum += 10`. Similarly, the expression `price *= rate` is equivalent to the expression `price = price * rate`.

In using these new assignment operators it is important to note that the variable to the left of the assignment operator is applied to the complete expression on the right. For example, the expression `price *= rate + 1` is equivalent to the expression `price = price * (rate + 1)`, not `price = price * rate + 1`.

Accumulating

Assignment expressions like `sum += 10` or its equivalent, `sum = sum + 10`, are very common in programming. These expressions are required in accumulating subtotals when data is entered one number at a time. For example, if we want to add the numbers 96, 70, 85, and 60 in calculator fashion, the following statements could be used:

Statement	Value in sum
`sum = 0;`	0
`sum = sum + 96;`	96
`sum = sum + 70;`	166
`sum = sum + 85;`	251
`sum = sum + 60;`	311

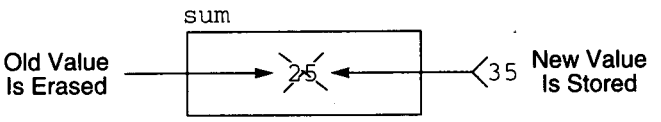


FIGURE 2–9 `sum = sum + 10;` Causes a New Value to be Stored in `sum`

The first statement initialized `sum` to `0`. This removes any number ("garbage" value) stored in `sum` that would invalidate the final total. As each number is added, the value stored in `sum` is increased accordingly. After completion of the last statement, `sum` contains the total of all the added numbers.

Program 2-5 illustrates the effect of these statements by displaying `sum`'s contents after each addition is made.

Program 2-5

```
#include <stdio.h>
main( )
{
  int sum;

  sum = 0;
  printf("\nThe value of sum is initially set to %d.", sum);
  sum = sum + 96;
  printf("\n  sum is now %d.", sum);
  sum = sum + 70;
  printf("\n  sum is now %d.", sum);
  sum = sum + 85;
  printf("\n  sum is now %d.", sum);
  sum = sum + 60;
  printf("\n  The final sum is %d.", sum);
}
```

The output displayed by Program 2-5 is:

```
The value of sum is initially set to 0.
   sum is now 96.
   sum is now 166.
   sum is now 251.
   The final sum is 311.
```

Although Program 2-5 is not a practical program (it is easier to add the numbers by hand), it does illustrate the subtotaling effect of repeated use of statements having the form

variable = variable + new_value;

We will find many uses for this type of statement when we become more familiar with the repetition statements introduced in Chapter 5.

Counting

An assignment statement that is very similar to the accumulating statement is the counting statement. Counting statements have the form

variable = variable + fixed_number;

Examples of counting statements are:

```
i = i + 1;
n = n + 1;
count = count + 1;
j = j + 2;
m = m + 2;
kk = kk + 3;
```

In each of these examples the same variable is used on both sides of the equal sign. After the statement is executed the value of the respective variable is increased by a fixed amount. In the first three examples the variables `i`, `n`, and `count` have all been increased by one. In the next two examples the respective variables have been increased by two, and in the final example the variable `kk` has been increased by three.

For the special case in which a variable is either increased or decreased by one, C provides two unary operators. Using the increment operator, `++`, the expression `variable = variable + 1` can be replaced by the expression `++variable`. Examples of the increment operator are:

Expression	Alternative
`i = i + 1`	`++i`
`n = n + 1`	`++n`
`count = count + 1`	`++count`

Program 2-6 illustrates the use of the increment operator.

Program 2-6

```
#include <stdio.h>
main( )
{
  int count;

  count = 0;
```

continued

```
  printf("\nThe initial value of count is %d.", count);
  ++count;
  printf("\n   count is now %d.", count);
  ++count;
  printf("\n   count is now %d.", count);
  ++count;
  printf("\n   count is now %d.", count);
  ++count;
}
```

The output displayed by Program 2-6 is:

```
The initial value of count is 0.
    count is now 1.
    count is now 2.
    count is now 3.
    count is now 4.
```

In addition to the increment operator, C also provides a decrement operator, `--`. As you might expect, the expression `--variable` is equivalent to the expression `variable = variable - 1`.

Examples of the decrement operator are:

Expression	Alternative
`i = i - 1`	`--i`
`n = n - 1`	`--n`
`count = count - 1`	`--count`

When `++` appears before a variable it is called a *prefix increment operator.* Besides appearing before (pre) a variable, the increment operator can also be applied after a variable; for example, in the expression `n++`. When the increment appears after a variable it is called a *postfix increment.* Both of these expressions, `++n` and `n++`, correspond to the longer expression `n = n + 1`. The distinction between a prefix and postfix increment operator occurs when the variable being incremented is used in an assignment expression. For example, the expression `k = ++n` does two things in one expression. Initially the value of `n` is incremented by one and then the new value of `n` is assigned to the variable `k`. Thus, the statement `k = ++n;` is equivalent to the two statements

```
n = n + 1;    /* increment n first      */
k = n;        /* assign n's value to k */
```

The assignment expression `k = n++`, which uses a postfix increment operator, reverses this procedure. A postfix increment operates after the assignment is

completed. Thus, the statement `k = n++;` first assigns the current value of `n` to `k` and then increments the value of `n` by one. This is equivalent to the two statements

```
k = n;        /* assign n's value to k */
n = n + 1;    /* and then increment n  */
```

Just as there are prefix and postfix increment operators, C also provides prefix and postfix decrement operators. For example, both of the expressions `--n` and `n--` reduce the value of `n` by one. These expressions are equivalent to the longer expression `n = n - 1`. As with the increment operator, however, the prefix and postfix decrement operators produce different results when used in assignment expressions. For example, the expression `k = --n` first decrements the value of `n` by one before assigning the value of `n` to `k`. But the expression `k = n--` first assigns the current value of `n` to `k` and then reduces the value of `n` by one.

The increment and decrement operators can often be used advantageously to reduce program storage requirements significantly and increase execution speed. For example, consider the following three statements:

```
count = count + 1;
count += 1;
++count;
```

All perform the same function; however, when these instructions are compiled for execution on an IBM personal computer the storage requirements for the executable instructions are 9, 4, and 3 bytes, respectively. Using the assignment operator, =, instead of the increment operator results in using three times the storage space for the instruction, with an accompanying decrease in execution speed.

Exercises 2.3

1. Write an assignment statement to calculate the circumference of a circle having a radius of 3.3 inches. The equation for determining the circumference, c, of a circle is $c = 2 \pi r$, where r is the radius and π equals 3.1416.

2. Write an assignment statement to calculate the area of a circle. The equation for determining the area, a, of a circle is $a = \pi r^2$, where r is the radius and $\pi = 3.1416$.

3. Write an assignment statement to convert temperature in degrees Fahrenheit to degrees Celsius. The equation for this conversion is Celsius = 5/9 (Fahrenheit − 32).

4. Write an assignment statement to calculate the round trip distance, d, in feet of a trip that is s miles long, one way.

5. Write an assignment statement to calculate the elapsed time in minutes, that it takes to make a trip. The equation for computing elapsed time is elapsed time = total distance / average speed. Assume that the distance is in miles and the average speed is in miles/hour.

6. Write an assignment statement to calculate the nth term in an arithmetic sequence. The formula for calculating the value, v, of the nth term is $v = a + (n-1)d$, where a = the first number in the sequence and d = the difference between any two numbers in the sequence.

7. Write an assignment statement to calculate the linear expansion in a steel beam as a

function of temperature increase. The formula for linear expansion, l, is $l = l_o[1+\alpha(T_f - T_o)]$, where l_o is the length of the beam at temperature T_o, α is the coefficient of linear expansion, and T_f is the final temperature of the beam.

8. Coulomb's law states that the force F acting between two electrically charged spheres is given by the formula $F = k\, q_1\, q_2 / r$, where q_1 is the charge on the first sphere, q_2 is the charge on the second sphere, r is the distance between the centers of the two spheres, and k is a proportionality constant. Write an assignment statement to calculate the force F.

9. Write an assignment statement to determine the maximum bending moment, M, of a beam. The formula for maximum bending moment is $M = X\, W\, (L - X) / L$, where X is the distance from the end of the beam that a weight, W, is placed, and L is the length of the beam.

10. Determine the output of the following program:

```
#include <stdio.h>
main( ) /* a program illustrating integer truncation */
{
  int num1, num2;

  num1 = 9/2;
  num2 = 17/4;
  printf("\nthe first integer displayed is %d", num1);
  printf("\nthe second integer displayed is %d", num2);
}
```

11. Determine the output produced by the following program:

```
#include <stdio.h>
main( )
{
  float average = 26.27;

  printf("\nthe average is %f", average);
  average = 682.3;
  printf("\nthe average is %f", average);
  average = 1.968;
  printf("\nthe average is %f", average);
}
```

12. Determine the output produced by the following program:

```
#include <stdio.h>
main( )
{
  float sum;

  sum = 0.0;
  printf("\nthe sum is %f", sum);
  sum = sum + 26.27;
  printf("\nthe sum is %f", sum);
  sum = sum + 1.968;
  printf("\nthe final sum is %f", sum);
}
```

13a. Determine what each statement causes to happen in the following program.

```
#include <stdio.h>
main( )
{
  int num1, num2, num3, total;

  num1 = 25;
  num2 = 30;
  total = num1 + num2;
  printf("\n %d + %d = %d", num1, num2, total);
}
```

b. What is the output that will be produced when the program listed in Exercise 13a is compiled and executed?

Note for Exercises 14 through 16: Identify the errors in the sections of code listed in each exercise:

14.

```
#include <stdio.h>
main( )
{
  width = 15
  area = length * width;
  printf("The area is %d",area
}
```

15.

```
#include <stdio.h>
main( )
{
  int length, width, area;

  area = length * width;
  length = 20;
  width = 15;
  printf("The area is %d",area);
```

16.

```
#include <stdio.h>
main( )
{
  int length = 20; width = 15, area;

  length * width = area;
  printf("The area is %d",area);
}
```

17. By mistake a student reordered the statements in Program 2-5 as follows:

```
#include <stdio.h>
main( )
{
  int sum;
```

```
    sum = 0;
    sum = sum + 96;
    sum = sum + 70;
    sum = sum + 85;
    sum = sum + 60;
    printf("\nThe value of sum is initially set to %d.", sum);
    printf("\n  sum is now %d.", sum);
    printf("\n  sum is now %d.", sum);
    printf("\n  sum is now %d.", sum);
    printf("\n  The final sum is %d.", sum);
}
```

Determine the output that this program produces.

18. Enter, compile, and execute Program 2-3 on your computer system.

19. Enter, compile, and execute Program 2-4 on your computer system.

20. Enter, compile, and execute Program 2-5 on your computer system.

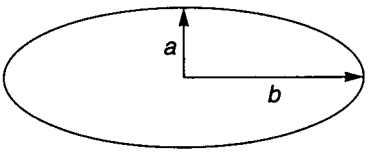
21. Using Program 2-3, determine the volume of cylinders having the following radii and heights:

Radius (in.)	Height (in.)
1.62	6.23
2.86	7.52
4.26	8.95
8.52	10.86
12.29	15.35

22. Modify Program 2-3 to calculate the weight in pounds of the steel cylinder whose volume was found by the program. The formula for determining the weight is *weight* = $.28\ \pi r^2 h$, where r is the radius in inches and h is the height in inches of the cylinder.

23. The area of an ellipse (see Figure 2–10) is given by the formula area = $\pi * a * b$. Using this formula, write a C program to calculate the area of an ellipse having a minor axis of 2.5 inches and a major axis of 6.4 inches.

FIGURE 2–10 The Minor Axis *a* and the Major Axis *b* of an Ellipse



24a. The combined resistance of three resistors connected in parallel, as shown in Figure 2–11, is given by the equation

$$\text{Combined resistance} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}$$

Write a C program to calculate and display the combined resistance when the three resistors $R_1 = 1000$, $R_2 = 1000$, and $R_3 = 1000$ are connected in parallel. Your program should produce the display:

```
The combined resistance, in ohms, is _____
```

where the underlined spaces are to be replaced by the value of the combined resistance computed by your program.

b. How do you know that the value calculated by your program is correct?

c. Once you have verified the output produced by your program, modify it to determine the combined resistance when the resistors $R_1 = 1500$, $R_2 = 1200$, and $R_3 = 2000$ are connected in parallel.

25a. Write a C program to calculate and display the value of the slope of the line connecting the two points whose coordinates are (3,7) and (8,12). Use the fact that the slope between two points having coordinates $(x1,y1)$ and $(x2,y2)$ is $(y2 - y1) / (x2 - x1)$.

b. How do you know that the result produced by your program is correct?

c. Once you have verified the output produced by your program, modify it to determine the slope of the line connecting the points (2,10) and (12,6).

26a. Write a C program to calculate and display the coordinates of the midpoint of the line connecting the two points given in Exercise 25a. Use the fact that the coordinates of the midpoint between two points having coordinates $(x1,y1)$ and $(x2,y2)$ are $((X1+X2)/2, (Y1+Y2)/2)$. Your program should produce the following display:

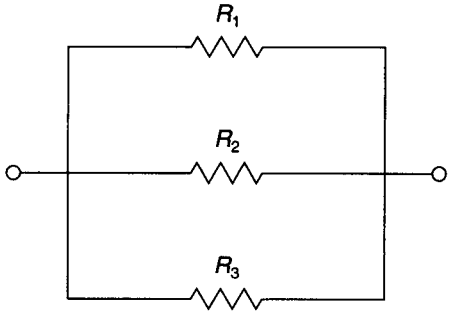
```
The x midpoint coordinate is _____
The y midpoint coordinate is _____
```

where the underlined spaces are to be replaced with the values calculated by your program.

b. How do you know that the midpoint values calculated by your program are correct?

c. Once you have verified the output produced by your program, modify it to determine the midpoint coordinates of the line connecting the points (2,10) and (12,6).

FIGURE 2–11 Three Resistors Connected in Parallel



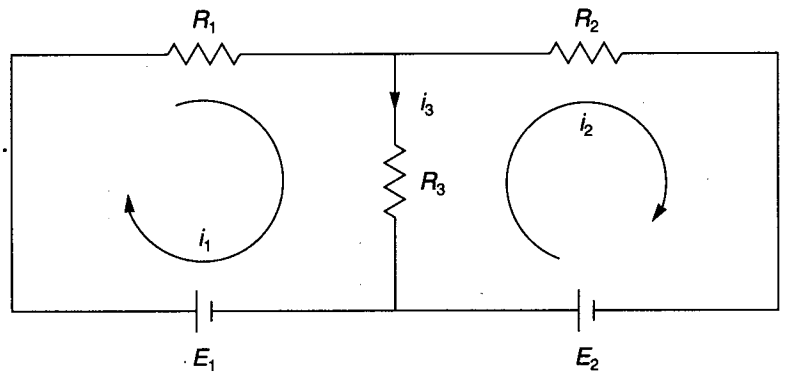


FIGURE 2–12 An Electrical Circuit

28a. For the electrical circuit shown in Figure 2–12, the currents I_1, I_2, and, I_3 can be determined using the formulas

$$I_1 = \frac{E_2R_3 + E_1(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$I_2 = \frac{E_1R_3 + E_2(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$I_3 = I_1 - I_2$$

Using these formulas write a C program to compute the currents when $R_1 = 10$ ohms, $R_2 = 4$ ohms, $R_3 = 6$ ohms, $E_1 = 12$ volts, and $E_2 = 9$ volts. The display produced by your program should be

```
Current i1 is _____
Current 12 is _____
Current i3 is _____
```

where the underlined spaces are to be replaced by the values determined in your program.

b. How do you know that the currents calculated by your program are correct?

c. Once you have verified the output produced by your program, modify it to determine the three currents for the following values: $R_1 = 1500$, $R_2 = 1200$, $R_3 = 2000$, $E_1 = 15$, and $E_2 = 12$.

2.4 Formatted Output

Besides displaying correct results, it is extremely important for a program to present its results attractively. Most programs are judged, in fact, on their perceived ease of data entry and the style and presentation of their output. For example, displaying a monetary result as `1.897000` is not in keeping with accepted report conventions. The display should be either `$1.90` or `$1.89`, depending on whether rounding or truncation is used.

The format of values displayed by `printf( )` can be controlled by *field width specifiers* included as part of each conversion control sequence. For example, the statement

```
printf("The sum of%3d and%4d is%5d.", 6, 15, 21);
```

causes the printout

```
The sum of  6 and  15 is   21.
```

The numbers `3`, `4`, and `5` in the control string are the field width specifiers. The `3` causes the first number to be printed in a total field width of three spaces, in this case two blank spaces followed by the number `6`. The field width specifier for the second conversion control sequence, `%4d`, causes two blank spaces and the number `15` to be printed for a total field width of four spaces. The last field width specifier causes the `21` to be printed in a field of five spaces, which includes three blanks and the number `21`. As illustrated, each integer is right-justified within the specified field.

Field width specifiers are useful in printing columns of numbers so that the numbers in each column align correctly. For example, Program 2-7 illustrates how a column of integers would align in the absence of field width specifiers.

Program 2-7

```
#include <stdio.h>
main()
{
   printf("\n%d", 6);
   printf("\n%d", 18);
   printf("\n%d, 124);
   printf("\n---");
   printf("\n%d", 6+18+124);
}
```

The output of Program 2-7 is:

```
 6
18
124
---
148
```

Since no field widths are given, the `printf( )` function allocates enough space for each number as it is received. To force the numbers to align on the units digit requires a field width wide enough for the largest displayed number. For Program 2-7, a width of three suffices. The use of this field width is illustrated in Program 2-8.

Program 2-8

```
#include <stdio.h>
main()
{
   printf("\n%3d", 6);
   printf("\n%3d", 18);
   printf("\n%3d, 124);
   printf("\n---");
   printf("\n%3d", 6+18+124);
}
```

The output of Program 2-8 is:

```
  6
 18
124
---
148
```

Formatted floating point numbers require the use of two field width specifiers. The first specifier determines the total width of the display, including the decimal point; the second determines how many digits are printed to the right of the decimal point. For example, the statement

```
printf("|%10.3f|",25.67);
```

causes the printout

```
|    25.670|
```

The bar character, `|`, is used to clearly mark the beginning and end of the display field. The field width specifier `10.3` tells `printf( )` to display the number in a total field of `10`, which includes one decimal point and three digits to the right of the decimal point. Since the number contains only two digits to the right of the decimal point, the decimal part of the number is padded with a trailing zero.

For all numbers (integers, floating point, and double precision), `printf( )` ignores the specified field width if the total field width is too small and allocates enough space for the integer part of the number to be printed. The fractional part of both floating point and double precision numbers is always displayed with the number of specified digits. If the fractional part contains fewer digits than specified, the number is padded with trailing zeros; if the fractional part contains more digits than called for in the specifier, the number is rounded to the indicated number of decimal places. Table 2–7 illustrates the effect of various field width specifiers.

TABLE 2–7 Effect of Field Width Specifiers

Specifier	Number	Display	Comments
`|%2d|`	3	`| 3|`	Number fits in field
`|%2d|`	43	`|43|`	Number fits in field
`|%2d|`	143	`|143|`	Field width ignored
`|%2d|`	2.3	Machine dependent	Floating point in an integer field
`|%5.2f|`	2.366	`| 2.37|`	Field of 5 with 2 decimal digits
`|%5.2f|`	42.3	`|42.30|`	Number fits in field
`|%5.2f|`	142.364	`|142.36|`	Field width ignored but fractional specifier used
`|%5.2f|`	142	Machine dependent	Integer in a floating point field

Format Modifiers

In addition to the conversion control sequences (`%d`, `%f`, etc.) and the field width specifiers that may be used with them, C also provides a set of format modifiers that provide additional format control, such as left and right field justification. Format modifiers, if used, must always be placed immediately after the `%` symbol. The more commonly used format modifiers are discussed here.

Left-Justification

Numbers displayed using the `printf( )` function are normally displayed right-justified with leading spaces inserted to fill the selected field width. To force the output to left-justify the display, a minus sign (-) format modifier can be used. For example, the statement

```
printf("|%-10d|",59);
```

causes the display

```
|59        |
```

Again, we have used the bar symbol, `|`, to clearly identify the beginning and end of the designated display. Notice that the displayed number, `59`, is printed at the beginning of the field (left-justification within the field) rather than at the end of the field, as would be obtained in the absence of the format modifier. Also notice that the format modifier within the `printf( )` function is placed immediately after the `%` symbol.

Explicit Sign Display

Normally, the sign of a number is only displayed for negative numbers. To force both positive and negative signs to be displayed, a plus sign (+) format modifier must be used. For example, the statement

```
printf("|%+10d|",59);
```

causes the display

```
|       +59|
```

In the absence of the plus sign immediately after the % symbol in the printf( ) function call, the output would not contain the sign of the positive number.

Format modifiers may be combined. For example, the conversion control sequence %-+10d causes an integer number to display its sign and to be left-justified in a field width of 10 spaces. As the order of the format modifiers is not critical, this conversion control sequence could just as well have been written %+-10d.

Other Number Bases[5]

When outputting integers several display conversions are possible. As we have seen, the conversion control sequence %d, with or without a field width specifier, causes integers to be displayed in decimal (base 10) form. To have the value of an integer displayed as either a base 8 (octal) or a base 16 (hexadecimal) number requires the use of the conversion control sequences %o and %x, respectively. Program 2-9 illustrates each of these conversion control sequences.

Program 2-9

```
#include <stdio.h>
main()  /* a program to illustrate output conversions */
{
  printf("The decimal (base 10) value of 15 is %d.", 15);
  printf("\nThe octal (base 8) value of 15 is %o.", 15);
  printf("\nThe hexadecimal (base 16) value of 15 is %x.", 15);
}
```

The output produced by Program 2-9 is:

```
The decimal (base 10) value of 15 is 15.
The octal (base 8) value of 15 is 17.
The hexadecimal (base 16) value of 15 is F.
```

[5] This topic may be omitted on first reading without loss of subject continuity.

The display of integer values in one of the three possible number systems (decimal, octal, and hexadecimal) does not affect how the number is actually stored inside a computer. All numbers are stored using the computer's own internal codes. The conversion control sequences convert the internal code for output display purposes.

Besides displaying integers in octal or hexadecimal form, integer constants can also be written in a program in these forms. To designate an octal integer constant, the number must have a leading zero. The number `023`, for example, is an octal number in C. Hexadecimal numbers are denoted using a leading `0x`. The use of octal and hexadecimal integer constants is illustrated in Program 2-10.

Program 2-10

```
#include <stdio.h>
main()
{
  printf("The decimal value of 025 is %d.\n",025);
  printf("The decimal value of 0x37 is %d.\n",0x37);
}
```

When Program 2-10 is run, the following output is obtained:

```
The decimal value of 025 is 21.
The decimal value of 0x37 is 55.
```

The relationship between the input, storage, and display of integers is illustrated in Figure 2–13.

To force both octal and hexadecimal numbers to be printed with a leading `0` and `0x`, respectively, the # format modifier must be used. For example, the statement

```
printf("The octal value of decimal 21 is %#o",21);
```

produces the display

```
The octal value of decimal 21 is 025
```

Without the inclusion of the # format modifier within the conversion control sequence `%0`, the displayed octal value would be 25, with no leading 0. Similarly, the statement

```
printf("The hexadecimal value of decimal 55 is %#x",55);
```

produces the display

```
The hexadecimal value of decimal 55 is 0x37
```

Without the inclusion of the # format modifier within the conversion sequence `%x`, the displayed hexadecimal value would be `37`, with no leading `0x`.

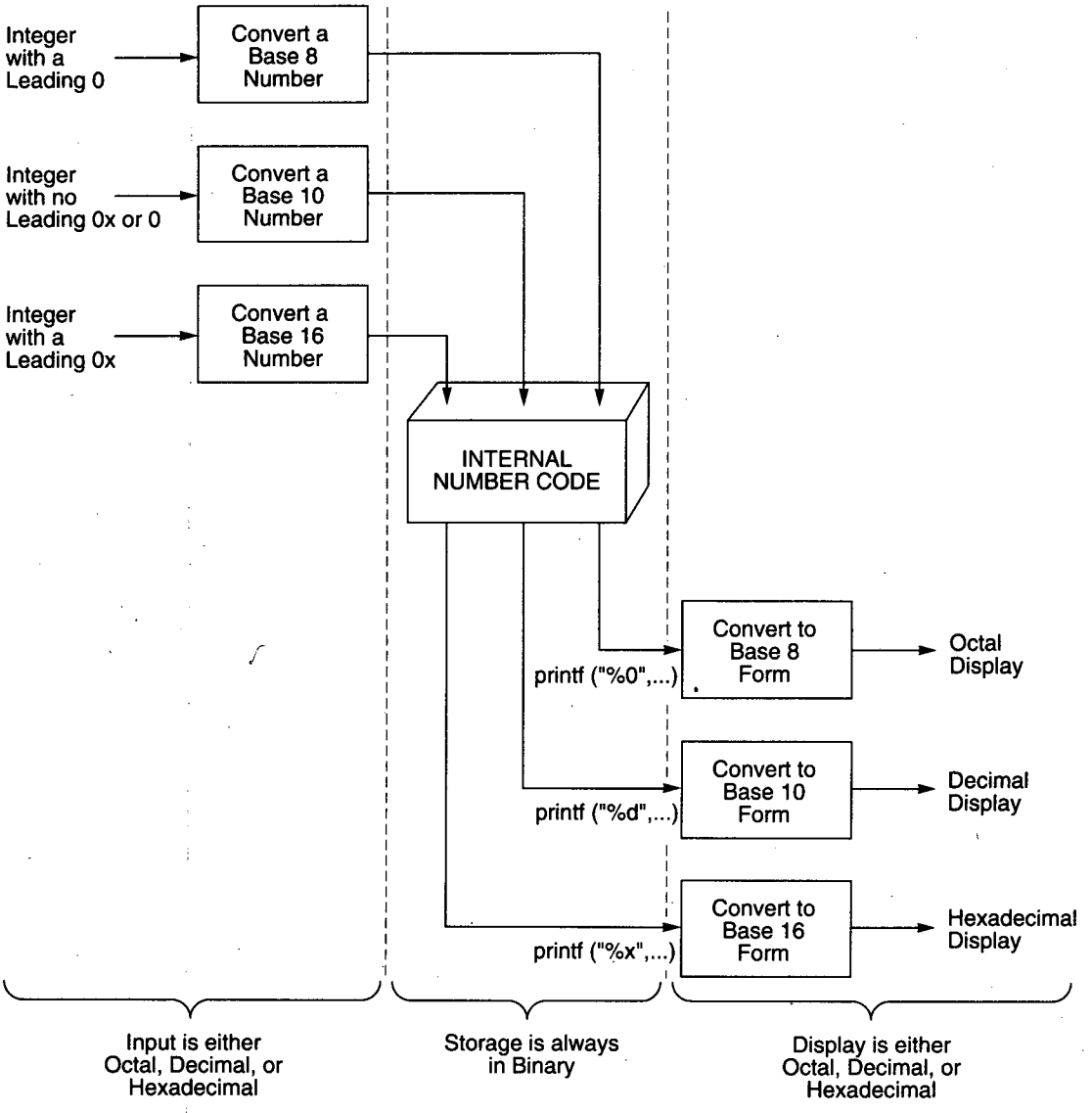


FIGURE 2–13 Input, Storage, and Display of Integers

Exercises 2.4

1. Determine and write out the display produced by the following statements:

a. `printf("|%d|",5);`
b. `printf("|%4d|",5);`
c. `printf("|%4d|",56829);`
d. `printf("|%5.2f|",5.26);`
e. `printf("|%5.2f|",5.267);`

f. `printf("|%5.2f|",53.264);`
g. `printf("|%5.2f|",534.264);`
h. `printf("|%5.2f|",534.);`

2. Write out the display produced by the following statements:

a.
```
printf("The number is %6.2f\n",26.27);
printf("The number is %6.2f\n",682.3);
printf("The number is %6.2f\n",1.968);
```

b.
```
printf("$%6.2f\n",26.27);
printf(" %6.2f\n",682.3);
printf(" %6.2f\n",1.968);
printf("--------\n");
printf("$%6.2f\n", 26.27 + 682.3 + 1.968);
```

c.
```
printf("$%5.2f\n",26.27);
printf(" %5.2f\n",682.3);
printf(" %5.2f\n",1.968);
printf("--------\n");
printf("$%5.2f\n", 26.27 + 682.3 + 1.968);
```

d.
```
printf("%5.2f\n",34.164);
printf("%5.2f\n",10.003);
printf("-----\n");
printf("%5.2f\n", 34.164 + 10.003);
```

3. Determine the errors in each of the following statements:

a. `printf("%d," 15)`
b. `printf("%f", 33);`
c. `printf("%5d", 526.768);`
d. `printf("a b c", 26, 15, 18);`
e. `printf("%3.6f", 47);`
f. `printf("%3.6", 526.768);`
g. `printf(526.768, 33,"%f %d");`

4a. Rewrite the `printf( )` function calls in the following program to produce the display:

```
The sales tax is $ 1.80
The total bill is $37.80
#include <stdio.h>
main()
{
  printf("The sales tax is %f", .05 * 36);
  printf("The total bill is %f", 37.80);
}
```

b. Run the program written for Exercise 4a to verify the output display.

5. Write a C program that displays the results of the expressions `(3.0 * 5.0)`, `(7.1 * 8.3 - 2.2)`, and `(3.2 / (6.1 * 5))` on three separate lines. Each value displayed should be limited to two decimal positions to the right of the decimal point. Calculate the value of these expressions manually to verify that the displayed values are correct.

6. The combined resistance of three resistors connected in parallel, as shown in Figure 2–14, is given by the equation

$$\text{Combined resistance} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}$$

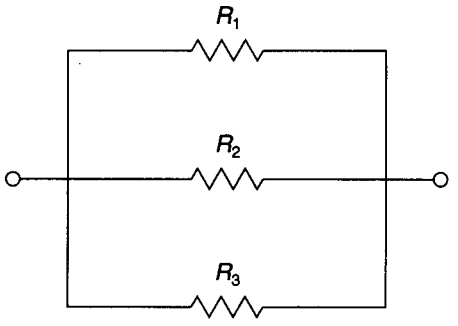


FIGURE 2–14 Three Resistors Connected in Parallel

Using this formula, write a C program to calculate and display the combined resistance when the three resistors $R_1 = 1000$, $R_2 = 1000$, and $R_3 = 1000$ are connected in parallel. The output should produce the display: `The combined resistance is xxxx.xx ohms`, where `xxxx.xx` denotes that the calculated value should be placed in a field width of seven columns with two positions to the right of the decimal point.

7. Write a C program to calculate and display the value of the slope of the line connecting the two points whose coordinates are (3,7) and (8,12). Use the fact that the slope between two points having coordinates ($x1$,$y1$) and ($x2$,$y2$) is $(y2 - y1) / (x2 - x1)$. The display produced by your program should be: `The value of the slope is xxx.xx`, where `xxx.xx` denotes that the calculated value should be placed in a field wide enough for three places to the left of the decimal point and two places to the right of it.

8. Write a C program to calculate and display the coordinates of the midpoint of the line connecting the two points given in Exercise 7. Use the fact that the coordinates of the midpoint between two points having coordinates ($x1$,$y1$) and ($x2$,$y2$) are ((X1+X2)/2, (Y1+Y2)/2). The display produced by your program should be:

```
The x coordinate of the midpoint is xxx.xx
The y coordinate of the midpoint is xxx.xx
```

where `xxx.xx` denotes that the calculated value should be placed in a field wide enough for three places to the left of the decimal point and two places to the right of it.

9. Write a C program to calculate and display the maximum bending moment, M, of a beam that is supported on both ends (see Figure 2–15). The formula for maximum bending moment is $M = X\,W\,(L - X) / L$, where X is the distance from the end of the beam that a weight, W, is placed and L is the length of the beam. Let W = 500 rhs, x = 10 ft. and L = 25 ft. The display produced by your program should be:

```
The maximum bending moment is xxxx.xxxx,
```

where `xxx.xxxx` denotes that the calculated value should be placed in a field wide enough for four places to the right and left of the decimal point.

10. For the electrical circuit shown in Figure 2–16, the currents, I_1, I_2, and I_3 can be determined using the formulas

$$I_1 = \frac{E_2R_3 + E_1(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

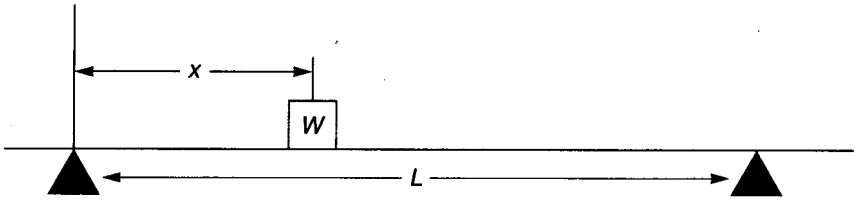


FIGURE 2–15 Calculating the Maximum Bending Moment

$$I_2 = \frac{E_1R_3 + E_2(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$I_3 = I_1 - I_2$$

Using these formulas write a C program to compute the three currents when R_1 = 1000 ohms, R_2 = 400 ohms, R_3 = 600 ohms, E_1 = 12 volts, and E_2 = 9 volts. The display produced by your program should be:

```
Current i1 is xx.xxxx
Current i2 is xx.xxxx
Current i3 is xx.xxxx
```

where xx.xxxx denotes that the calculated value should be placed in a field wide enough for two places to the left of the decimal point and four places to the right of it.

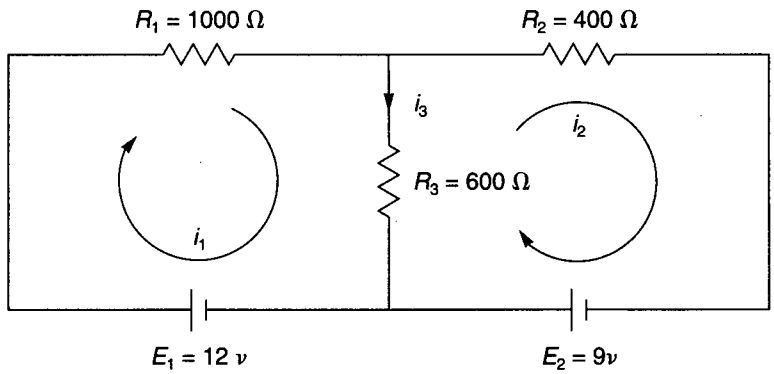


FIGURE 2–16 Calculating Currents in an Electrical Circuit

2.5 Top-Down Program Development

Recall from Section 1.1 that writing a C program is essentially the last step in the programming process. The first step in the process is determining what is required and selecting the algorithm to be coded into C. In this section we present a five-step program development procedure called top-down development for converting programming problems into working C programs. To make this development procedure more meaningful, we first apply it to a simple program-

ming problem. As we will see, designing a program using a top-down approach results in a modular program design.

The five steps in the top-down development procedure are:

1. Determine the desired output items that the program must produce.
2. Determine the input items.
3. Design the program as follows:
   a. Select an algorithm for transforming the input items into the desired outputs.
   b. Check the chosen algorithm by hand, using specific input values.
   c. Determine variable names for the selected algorithm
4. Code the algorithm into C.
5. Test the program using selected test data.

Steps 1 and 2 in the development procedure are referred to as the program *Analysis Phase,* Step 3 is called the *Design Phase,* Step 4 the *Coding Phase,* and Step 5 the *Testing Phase.*

In the analysis phase of program development (Steps 1 and 2) we are concerned with extracting the complete input and output information supplied by the problem. Together these two items are referred to as the problem's input/output, or I/O for short. Only after a problem's I/O has been determined is it possible to select an algorithm for transforming the inputs into the desired outputs. For example, consider the following simple programming problem:

> The electrical resistance of a metal wire, in ohms, is given by the formula $r = (ml)/a$ where m is the resistivity of the metal; l is the length of the wire in feet; and a is the cross-sectional area of the wire in circular mils. Using this information, write a C program to calculate the resistance of a wire that is 125 feet long, has a cross-sectional area of 500 circular mils, and is copper. The resistivity of copper is 10.4.

Step 1: Determine the Desired Output

The first step in developing a program for this problem statement is to determine the required outputs (Step 1 of the development procedure). Frequently, the statement of the problem will use such words as *calculate, print, determine, find,* or *compare,* which can be used to determine the desired outputs.

For our sample problem statement, the key phrase is "to calculate the resistance of a wire." This clearly identifies an output item. Since there are no other such phrases in the problem, only one output item is required.

Step 2: Determine the Input Items

After we have clearly identified the desired output, Step 2 of the development process requires that we identify all input items. It is essential at this stage to distinguish between input items and input values. An input item is the name of an input quantity, while an input value is a specific number or quantity that the input item can be. For example, in our sample problem statement the input items are the resistivity, m, the length of the wire, l, and the cross-sectional area of the wire, a. Although these input items have specific numerical values, these input item values are generally not important in Step 2.

The reason that input values are not needed at this point is that the initial selection of an algorithm is typically independent of specific input values; the algorithm depends on knowing what the output and input items are and if there are any special limits. Let us see why this is so as we determine a suitable algorithm for our sample problem statement.

Step 3a: Determine an Algorithm

From the problem statement it is clear that the algorithm for transforming the input items to the desired output is given by the formula $r = (ml)/a$. Notice that this formula can be used regardless of the specific values assigned to m, l, and a. Although we cannot produce an actual numerical value for the output item, resistance, unless we have actual numerical values for the input items, the correct relationship between inputs and outputs is expressed by the formula. Recall that this is precisely what an algorithm provides: a description of how the inputs are to be transformed into outputs that works for all inputs. Thus, the complete algorithm, in pseudocode, for solving this problem is:

Assign values to m, l, and a
Calculate the resistance using the formula r=(ml)/a
Display the result

Step 3b: Do a Hand Calculation

After an algorithm has been selected, the next step in the design procedure, Step 3b, is to check the algorithm manually using specific data. Performing a manual calculation, either by hand or using a calculator, helps to ensure that you really do understand the problem. An added feature of doing a manual calculation is that the results can be used later to verify the operation of your program in the testing phase. Then, when the final program is used with other data, you will have established a degree of confidence that a correct result is being calculated.

Doing a manual calculation requires that we have specific input values that can be applied to the algorithm to produce the desired output. For this problem three input values are given: a resistivity of 10.4, a cross-sectional area of 500 circular mils, and a length of 125 feet. Substituting these values into the formula, we obtain a resistance of 2.60 ohms for the copper wire.

Step 3c: Select Variable Names

The last step in the design phase (Step 3c) is to choose the names of variables to hold the input, output, and any intermediate calculated items determined in the analysis phase (Steps 1 and 2). Let us use the variables named `restvy`, `area`, and `length` for the input items resistivity, area, and length, respectively; and a variable named `resist` for the calculated output, the resistance of the wire. All of these names are arbitrary and any valid symbolic names can be used in their place.

Step 4: Write the Program

Since we have selected variable names for the chosen algorithm, all that is required is for our program to declare these variables; initialize the input variables appropriately; compute the resistance variable; and print the calculated resistance value. Program 2-11 performs these steps.

Program 2-11

```
#include <stdio.h>
main( )
{
  float restvy, area, length, resist;
    .
  restvy = 10.4;
  area = 500;
  length = 125;
  resist = (restvy * length) / area;
  printf("\n The resistance of the wire (in ohms) is %f", resist);
}
```

When program 2-11 is executed, the following output is produced:

```
The resistance of the wire (in ohms) is 2.600000
```

Now that we have a working program that produces a result, the final step in the development process, testing the program, can begin.

Step 5: Test the Output

The purpose of testing is to verify that a program works correctly and actually fulfills its requirements. Once testing has been completed the program can be used to calculate outputs for differing input data without the need for retesting. This is, of course, one of the real values in writing a program: the same program can be used over and over with new input data.

In theory, testing would reveal all existing program errors (in computer terminology, a program error is called a *bug*). In practice, this would require checking all possible combinations of statement execution. Because of the time and effort required, this is usually an impossible goal except for extremely simple programs such as Program 2-11. (We illustrate why this is generally an impossible goal in Chapter 4, which describes C's IF statements.)

The inability to completely test most programs has led to various testing methodologies. The simplest of these methods is to verify the program's operation for carefully selected sets of input data. One set of input data that should always be used is the data that was selected for the hand calculation made previously in Step 3b of the development procedure. If testing reveals an error (bug), the process of debugging, which includes locating, correcting, and verifying the correction, can be initiated. It is important to realize that although this type of verification testing may reveal the presence of an error, it does not necessarily indicate the absence of one. Thus, the fact that a test does not reveal an error does not indicate that another bug is not lurking somewhere else in the program.

Modularity and Top-Down Design

The design of Program 2-11 was relatively simple. For more complex problems the design of a program's structure can be considerably more involved. In its

more elaborate form, designing a program is similar to receiving the pieces of a puzzle (the inputs) and deciding how to arrange them to form a completed structure (the desired output). Unlike a jigsaw puzzle, however, the pieces of a program design puzzle can be arranged in many different ways depending on the algorithm chosen for transforming the inputs into the desired outputs. In this regard, the program designer is similar to an architect who must draw up the plans for a house.

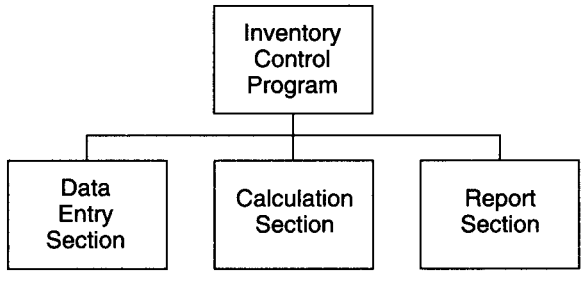
The general procedure for designing programs (Step 3 in our development procedure) is called *top-down design*. The purpose of top-down design is to design an algorithm that results in a modular program structure. To achieve this goal the design starts from the highest level requirement and proceeds to the parts that must be constructed to achieve this requirement. To make this more meaningful, consider that an inventory reporting program is required to keep track of the number of parts in inventory. The required output for this program is a description of all parts carried in inventory and the number of units of each item in stock; the given inputs are the initial inventory quantity of each part, the number of items sold, the number of items returned, and the number of items purchased.

For these I/O specifications, a designer could initially organize the requirements for the program into the three sections illustrated in Figure 2–17. This is called a *first-level structure diagram* because it represents the first overall structure of the program selected by the designer.

In top-down design the lower boxes in the structure diagram are refined until the tasks indicated in the boxes are small enough to be programmed as individual program units. For example, both the data entry and report subsections shown in Figure 2–17 would be further refined into suitable segments. The data entry section certainly must include provisions for entering the data. Since it is the system designer's responsibility to plan for contingencies and human error, provisions must also be made for handling incorrect data after an entry has been made and for deleting a previously entered value altogether. Similar subdivisions for the report section can also be made. Figure 2–18 illustrates a second-level structure diagram for an inventory tracking system that includes these further refinements.

The process of refinement continues until the last level of tasks can be coded using individual program units. Notice that the design produces a tree-like structure where the levels branch out as we move from the top of the structure to the bottom. When the design is complete it specifies both how many program units are needed and the calling sequence of each unit (that is, lower level units are

FIGURE 2–17 First-Level Structure Diagram



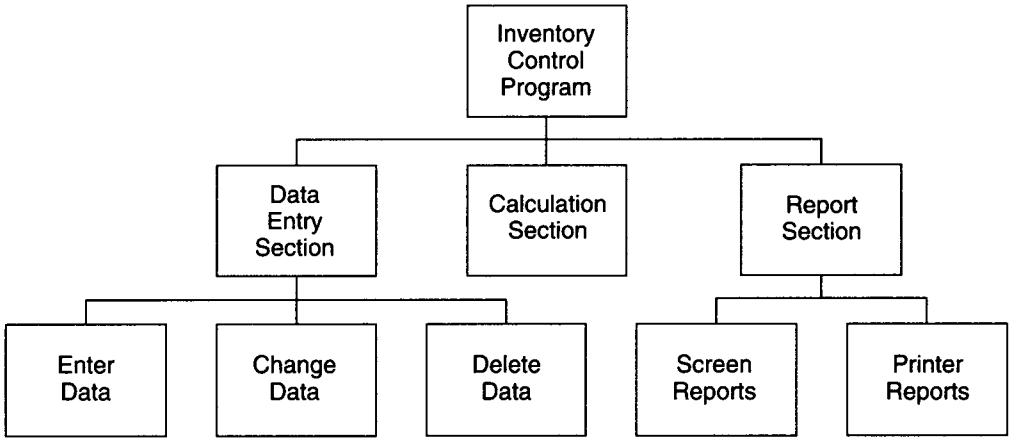


FIGURE 2–18 Second-Level Refinement Structure Diagram

called from higher level ones). The individual algorithms specified for each box on the final structure diagram, which are coded using separate functions, are frequently described using either flowcharts or pseudocode.

Exercises 2.5

Note: ***In each of these exercises a programming problem is given. Read the problem statement first and then answer the questions pertaining to the problem. Do not write the program.***

1. Consider the following programming problem: A C program is required that calculates the amount, in dollars, contained in a piggybank. The bank contains half-dollars, quarters, dimes, nickels, and pennies.

a. For this programming problem how many outputs are required?
b. How many inputs does this problem have?
c. Determine an algorithm for converting the input items into output items.
d. Test the algorithm written for part c using the following sample data: half-dollars = 0, quarters = 17, dimes = 24, nickels = 16, pennies = 12.

2. Consider the following programming problem: A C program is required to calculate the value of distance, in miles, given the relationship

$$\text{distance} = \text{rate} * \text{elapsed time}$$

a. For this programming problem how many outputs are required?
b. How many inputs does this problem have?
c. Determine an algorithm for converting the input items into output items.
d. Test the algorithm written for part c using the following sample data: rate is 55 miles per hour and elapsed time is 2.5 hours.
e. How must the algorithm you determined in part c be modified if the elapsed time is given in minutes instead of hours?

3. Consider the following programming problem: A C program is required to determine the value of Ergies, given the relationships

$$\text{Ergies} = \text{Fergies} * \sqrt{\text{Lergies}}$$
$$\text{Lergies} = 2P$$

a. For this programming problem how many outputs are required?
b. How many inputs does this problem have?
c. Determine an algorithm for converting the input items into output items.
d. Test the algorithm written for part c using the following sample data: Fergies = 14.65, and P = 8.

4. Consider the following programming problem: A C program is required to display the following name and address:

Mr. J. Knipper
63 Seminole Way
Dumont, N.J. 07030

a. For this program problem how many lines of output are required?
b. How many inputs does this problem have?
c. Determine an algorithm for converting the input items into output items.

5. Consider the following program problem: A C program is required to determine how far a car has traveled after 10 seconds, assuming the car is initially traveling at 60 miles per hour and the driver applies the brakes to uniformly decelerate at a rate of 12 miles/sec^2. Use the fact that distance = $s - (1/2)dt^2$, where s is the initial speed of the car, d is the deceleration, and t is the elapsed time.
a. For this programming problem how many outputs are required?
b. How many inputs does this problem have?
c. Determine an algorithm for converting the input items into output items.
d. Test the algorithm written for part c using the data given in the problem.

6. Consider the following programming problem: In 1627, Manhattan Island was sold to the Dutch settlers for approximately $24. If the proceeds of that sale had been deposited in a Dutch bank paying 5 percent interest, compounded annually, what would the principal balance be at the end of 1990? A display is required as follows:

```
Balance as of December 31, 1990 is: _______
```

where the underlined blank spaces are to be filled in by the amount calculated by your program.
a. For this programming problem how many outputs are required?
b. How many inputs does this problem have?
c. Determine an algorithm for converting the input items into output items.
d. Test the algorithm written for part c using the data given in the problem statement.

7. Consider the following programming problem: A C program is required that calculates and displays the weekly gross pay and net pay of two individuals. The first individual is paid an hourly rate of $8.43 and the second individual is paid an hourly rate of $5.67. Both individuals have 20 percent of their gross pay withheld for income tax purposes and both pay 2 percent of their gross pay, before taxes, for medical benefits.
a. For this programming problem how many outputs are required?
b. How many inputs does this problem have?
c. Determine an algorithm for converting the input items into output items.
d. Test the algorithm written for part c using the following sample data: The first person works 40 hours during the week and the second person works 35 hours.

8. Consider the following programming problem: The formula for the standard normal deviate, z, used in statistical applications is:

$$z = \frac{X - \mu}{\sigma}$$

where μ refers to a mean value and σ to a standard deviation. Using this formula, A C program is required that calculates and displays the value of the standard normal deviate when $X = 85.3$, $\mu = 80$, and $\sigma = 4$.

a. For this programming problem how many outputs are required?
b. How many inputs does this problem have?
c. Determine an algorithm for converting the input items into output items.
d. Test the algorithm written for part c using the data given in the problem.

9. Consider the following programming problem: The equation of the normal (bell-shaped) curve used in statistical applications is:

$$y = \frac{1}{\sigma\sqrt{2\pi}} e^{-[(\frac{1}{2})(x-u)/\sigma]^2}$$

Using this equation, A C program is required to calculate the value of y.

a. For this programming problem how many outputs are required?
b. How many inputs does this problem have?
c. Determine an algorithm for converting the input items into output items.
d. Test the algorithm written for part c using assuming $\mu = 90$, $\sigma = 4$, $x = 80$, e = 2.71828 and $\pi = 3.1416$.

2.6 Applications

In this section we apply the top-down development procedure presented in the previous section to two specific applications. Although each application is different, the top-down development procedure can be applied to any programming problem to produce a completed program.

Application 1: Pendulum Clocks

Pendulums used in clocks keep relatively accurate time for the following reason: When the length of a pendulum is relatively large compared to the maximum arc of its swing the time to complete one swing is independent of both the pendulum's weight and the maximum displacement of the swing. When this condition is satisfied the relationship between the time to complete one swing and the length of the pendulum is given by the formula

$$\text{length} = g\,[\text{time}/(2\,\pi)]^2$$

where π, accurate to four decimal places, is equal to 3.1416 and g is the gravitational constant equal to 32.2 ft/sec^2. When the time of a complete swing is given in seconds, the length of the pendulum is in feet. Using the given formula, write a C program to calculate and display the length of a pendulum needed to produce a swing that will be completed in one second. The length should be displayed in inches.

Program Development

Using our five-step development procedure we have:

Step 1: Determine the Required Outputs For this problem a single output is required by the program: the length of the pendulum. The problem also specifies that the actual value be displayed in units of inches.

Step 2: Determine All Input Items The input items required for this problem are the time to complete one swing, the gravitational constant, g, and pi.

Step 3: Design the Program

a. The algorithm for transforming the three input items into the desired output item is given by the formula length = g [time/(2 π)]^2. Since this formula calculates the length in feet, we will have to multiply the result by 12 to convert the answer into inches.

b. A hand calculation, using the data that g = 32.2, time = 1, and π = 3.1416, yields a length of 9.78 inches for the pendulum.

c. We select the variable names `time` for the time and `length` for the length. As g and pi are constants that do not change, we will not assign them to variables; instead, their values will be directly incorporated in the assignment statement used to determine the pendulum's length.

Step 4: Write the Program Program 2-12 provides the necessary code.

Program 2-12

```
#include <stdio.h>
main( )
{
  float time, length;

  time = 1.0;
  length = 12.0 * 32.2 * time /(2.0*3.1416) * time /(2.0*3.1416);
  printf("\n The length is %4.2f inches.", length);
}
```

Program 2-12 begins with a `#include` preprocessor command followed by a `main( )` function. This function starts with the reserved word `main` and ends with the closing brace, `}`. Additionally, Program 2-12 contains one declaration statement, two assignment statements, and one output statement. The assignment statement `time = 1.0;` is used to initialize the `time` variable. The assignment statement

```
length = 12.0 * 32.2 * time /(2.0*3.1416) * time/(2.0*3.1416);
```

calculates a value for the variable length. Notice that the `12.0` is used to convert the calculated value from feet into inches. Also notice the placement of parentheses in the expression `time/(2.0*3.1416)`. The parentheses ensures that the value of π is multiplied by `2.0` before the division is performed. If these parentheses were not included, the value of `time` would first be divided by `2.0`, and then the quantity `time/2.0` would be multiplied by π. Finally, this same quantity is multiplied by itself to obtain the necessary squared value (in the next chapter we will see how to use C's power function to obtain the same result). When Program 2-12 is compiled and executed the following output is produced:

```
The length is 9.78 inches.
```

Step 5: Test the Program The last step in the development procedure is to test the output of the program. As the displayed value agrees with the previous hand calculation, we have established a degree of confidence in the program. This permits us to use the program for different values of time. Note that if the parentheses were not correctly placed in the assignment statement that calculated a value for `length`, the displayed value would not agree with our previous hand calculation. This would have alerted us to the fact that there was an error in the program.

Application 2: Telephone Switching Networks

A directly connected telephone network is one in which all telephones in the network are directly connected and do not require a central switching station to establish calls between two telephones. For example, financial institutions on Wall Street use such a network to maintain direct and continuously open phone lines between institutions.

The number of direct lines, l, needed to maintain a directly connected network for n telephones is given by the formula

$$l = n(n-1)/2$$

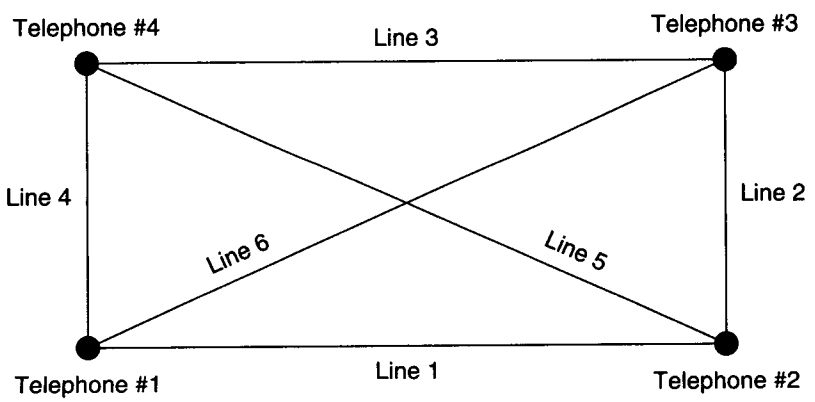
For example, directly connecting 4 telephones without the use of a central switching station requires 6 individual lines (see Figure 2–19). Adding a fifth telephone to the network illustrated in Figure 2–19 would require an additional 4 lines for a total of 10 lines.

Using the given formula write a program that determines the number of direct lines required for 100 subscribers, and the additional lines required if 10 new subscribers were added to the network.

Program Development

Using our five-step development procedure we have:
Step 1: Determine the Required Outputs For this program two outputs are required: the number of direct lines for 100 telephones and the additional number of lines needed when 10 new telephones are added into the existing network.
Step 2: Determine All Input Items The input items required for this problem are the number of telephones, denoted as n in the formula.

FIGURE 2–19 Directly Connecting Four Telephones



Step 3: Design the Program

a. The first output is easily obtained using the formula $l = n(n-1)/2$. Although there is no formula given for additional lines, we can use the given formula to determine the total number of lines needed for 110 telephones. Subtracting the number of lines for 100 telephones from the number of lines needed for 110 telephones yields the number of additional lines required. Thus, the complete algorithm for our program, in pseudocode, is:

Calculate the number of direct lines for 100 telephones
Calculate the number of direct lines for 110 telephones
Calculate the additional lines needed, which is the
difference between the second and first calculations
Display the number of lines for 100 telephones
Display the additional lines needed

b. A hand calculation using the data given yields:

$$l = 100(100 - 1)/2 = 100(99)/2 = 4950 \text{ lines}$$

for 100 telephones and that $l = 5995$ direct lines are needed for 110 telephones. Thus, an additional 1045 lines would be needed to directly connect the 10 additional telephones into the existing network.

c. We select the variable name `numin` for the initial number of 100 telephones, the variable name `numfin` for the final number of 110 telephones, `lines1` for the initial number of lines, and `lines2` for the final number of lines.

Step 4: Write the Program Program 2-13 provides the necessary code.

Program 2-13

```
#include <stdio.h>
main( )
{
  int numin, numfin, lines1, lines2;

  numin = 100;
  numfin = 110;
  lines1 = numin * (numin - 1)/2;
  lines2 = numfin * (numfin - 1)/2;
  printf("\nThe number of initial lines is %d", lines1);
  printf("\nThere are %d additional lines needed.", lines2 - lines1);
}
```

As before, the C program begins with the reserved word `main` and ends with the closing brace, `}`. Since the number of lines between telephones must be an integer (a fractional line is not possible), the variables `lines1` and `lines2` are specified as integer variables. The first two assignment statements initialize the

variables numin and numfin. The next assignment statement calculates the number of lines needed for 100 telephones and the last assignment statement calculates the number of lines for 110 telephones. The first call to printf( ) is used to display a message and the result of the first calculation. The second call to printf( ) is used to display the difference between the two calculations. The following output is produced when Program 2-13 is compiled and executed:

```
The number of initial lines is 4950
There are 1045 additional lines needed.
```

Additional Exercises for Chapter 2

1a. Modify Program 2-12 to calculate the length of a pendulum that produces an arc that takes two seconds to complete.
b. Compile and execute the program written for Exercise 1a on a computer.

2a. Modify Program 2-12 to determine the time it takes a three-foot pendulum to complete one swing. Your program should produce the following display:

```
The time to complete one swing (in seconds) is: _____
```

where the underlined blank spaces are replaced by the actual value calculated by your program.
b. Compile and execute the program written for Exercise 2a on a computer. Make sure to do a hand calculation so that you can verify the results produced by your program.
c. After you have verified the results of the program written in Exercise 2a, modify the program to calculate the time it takes a four-foot pendulum to complete one swing.

3a. Modify Program 2-13 to calculate and display the total number of lines needed to directly connect 1000 individual phones to each other.
b. Compile and execute the program written for Exercise 3a on a computer.

4a. Modify Program 2-13 so that the variable numfin is initialized to 10, which is the additional number of subscribers to be connected to the existing network. Make any other changes in the program so that the program produces the same display as Program 2-13.
b. Compile and execute the program written for Exercise 4a on a computer. Check that the display produced by your program matches the display shown in the text.

5a. Write, compile, and execute a C program to convert temperature in degrees Fahrenheit to degrees Celsius. The equation for this conversion is Celsius = 5.0/9.0 (Fahrenheit − 32.0). Have your program convert and display the Celsius temperature corresponding to 98.6 degrees Fahrenheit. Your program should produce the display:

```
For a fahrenheit temperature of _____ degrees
The equivalent celsius temperature is _____ degrees
```

where appropriate values are inserted by your program in place of the underlined blank spaces.
b. Check the values computed by your program by hand. After you have verified that your program is working correctly, modify it to convert 86.5 degrees Fahrenheit into its equivalent Celsius value.

6a. Write, compile, and execute a C program to calculate the dollar amount contained in

a piggy bank. The bank currently contains 12 half-dollars, 20 quarters, 32 dimes, 45 nickels, and 27 pennies. Your program should produce the following display:

```
The value of money, in dollars, is xx.xx
```

where `xx.xx` is replaced by the actual value calculated by your program and represents a total field width of 5 places with 2 places to the right of the decimal point.
b. Check the values computed by your program by hand. After you have verified that your program is working correctly, modify it to determine the dollar value of a bank containing no half-dollars, 17 quarters, 19 dimes, 10 nickels, and 42 pennies.

7a. Write, compile, and execute a C program to calculate the elapsed time it took to make a 183.67 mile trip. The equation for computing elapsed time is elapsed time = total distance / average speed. Assume that the average speed during the trip was 58 miles per hour. Your program should produce the display

```
The time for the trip was xx.xx hours
```

where `xx.xx` is replaced by the actual value calculated by your program and represents a total field width of 5 places with 2 places to the right of the decimal point.
b. Check the values computed by your program by hand. After you have verified that your program is working correctly, modify it to determine the elapsed time it takes to make a 372 mile trip at an average speed of 67 miles/hour.

8a. Write, compile, and execute a C program to calculate the sum of the numbers from 1 to 100. The formula for calculating this sum is sum = $(n/2)$ (2*a + $(n-1)d$), where n = number of terms to be added, a = the first number, and d = the difference between each number. Your program should produce the display

```
The sum of the numbers is xxxx
```

where the `xxxx` represents a field width of four digits and should be replaced by the sum computed by your program.
b. Check the values computed by your program by hand. After you have verified that your program is working correctly, modify it to determine the sum of the integers from 100 to 1000.

Note: Exercises 9 through 13 require raising a number to a power. This can be accomplished using C's power function `pow( )`***. For example, the statement*** `pow(2.0,5.0);` ***raises the number*** `2.0` ***to the 5th power, and the statement*** `pow(num1,num2);` ***raises the variable*** `num1` ***to the*** `num2` ***power. To use the power function the declaration statement*** `double pow( );` ***should be included with the variable declaration statements used in your program. The power function is explained in more detail in Section 3.1.***

9a. Newton's law of cooling states that when an object with an initial temperature T is placed in a surrounding substance of temperature A, it will reach a temperature *TFIN* in t minutes according to the formula

$$TFIN = (T - A)\, e^{-kt} + A$$

In this formula e is the irrational number 2.71828 rounded to five decimal places, commonly known as Euler's number, and k is a thermal coefficient, which depends on the material being cooled. Using this formula write, compile, and execute a C program that determines the temperature reached by an object after 20 minutes when it is placed

in a glass of water whose temperature is 60 degrees. Assume that the object initially has a temperature of 150 degrees and has a thermal constant of 0.0367. Your program should produce the display

```
The final temperature is xxx.xx
```

where xxx.xx represents a total field width of six digits, which includes a leading sign and two digits to the right of the decimal point. The value placed in this field should be calculated by your program.

b. Check the value computed by your program by hand. After you have verified that your program is working correctly, modify the object to determine the temperature reached after 10 minutes when the object is placed in a glass of water whose temperature is 50 degrees.

10a. Given an initial deposit of money, denoted as A, in a bank that pays interest annually, the amount of money at a time N years later is given by the formula

$$Amount = A * (1 + I)^N$$

where I is the interest rate as a decimal number (e.g., 9.5 percent is .095). Using this formula write, compile, and execute a C program that determines the amount of money that will be available in 4 years if $10,000 is deposited in a bank that pays 10 percent interest annually. Your program produce the display

```
The value after xx years is yyyy.yy
```

where the xs represent a field width of two and the ys represent a field width of seven, with two digits to the right of the decimal point. The xs are to be replaced by the number of years and the ys by the value of money calculated by your program. (See the note prior to Exercise 9 for information on C's power function.)

b. Check the value computed by your program by hand. After you have verified that your program is working correctly, modify it to determine the amount of money available if $24 dollars is invested at 4 percent for 300 years. Increase the field sizes of your output statement to accommodate both the number of years and the final amount.

11a. The present value of a dollar amount is the amount of money that must be deposited in a bank account today to yield a specified dollar amount in the future. For example, if a bank is currently paying 8 percent interest annually, you would have to deposit $6,947.90 in the bank today to have $15,000 in 10 years. Thus, the present value of the $15,000 is $6,947.90. Using this information, write, compile, and execute a C program that calculates how much must be deposited in a bank today to provide exactly $8,000 in 9 years at an annual interest rate of 8 percent. Use the formula

Present value = Future amount / (1.0 + annual interest rate)$^{\text{Years}}$
(See the note prior to Exercise 9 for information on C's power function.)

b. Check the value computed by your program by hand. After you have verified that your program is working correctly, modify it to determine the amount of money that must be invested in a bank today to yield $15,000 in 18 years at an annual rate of 6 percent.

12a. The set of linear equations

$$a_{11}x_1 + a_{12}x_2 = c_1$$
$$a_{21}x_1 + a_{22}x_2 = c_2$$

can be solved using Cramer's rule as

$$x_1 = \frac{c_1 a_{22} - a_{12} c_2}{a_{11} a_{22} - a_{12} a_{21}}$$

$$x_2 = \frac{a_{11} c_2 - c_1 a_{21}}{a_{11} a_{22} - a_{12} a_{21}}$$

Using these equations, write, compile, and execute a C program to solve for the x_1 and x_2 values that satisfy the following equations:

$$3x_1 + 4x_2 = 40$$
$$5x_1 + 2x_2 = 34$$

b. Check the values computed by your program by hand. After you have verified that your program is working correctly, modify it to solve the following set of equations:

$$3\,x_1 + 12.5\,x_2 = 22.5$$
$$4.2\,x_1 - 6.3\,x_2 = 30$$

2.7 Common Programming Errors

The common programming errors associated with the material presented in this chapter are:

1. Forgetting to declare all the variables used in a program. This error is detected by the compiler and an error message is generated for all undeclared variables.
2. Storing an incorrect data type in a declared variable. This error is detected by the compiler and the assigned value is converted to the data type of the variable it is assigned to. However, assigning a floating point value to an integer variable will result in the loss of the fractional part.
3. Using a variable in an expression before a value has been assigned to the variable. Here, whatever value happens to be in the variable will be used when the expression is evaluated, and the result will be meaningless.
4. Dividing integer values incorrectly. This error is usually disguised within a larger expression and can be very troublesome to detect. For example, the expression

```
3.425 + 2/3 + 7.9
```

yields the same result as the expression

```
3.425 + 7.9
```

because the integer division of 2/3 is 0.

5. Mixing data types in the same expression without clearly understanding the effect produced. Since C allows expressions with "mixed" data types, it is important to be clear about the order of evaluation and the data type of all intermediate calculations. As a general rule it is better never to mix data types in an expression unless a specific effect is desired.
6. Not including the correct number and type of conversion control sequences in `printf( )` function calls for the data types of the remaining arguments.
7. Not closing the control string in `printf( )` with a double quote symbol " followed by a comma when additional arguments are passed to `printf( )`.
8. Forgetting to separate all arguments passed to `printf( )` with commas.

2.8 Chapter Summary

1. Four types of data constants were introduced in this chapter: integer, floating point, double precision, and character values. Each of these types of data is typically stored in a computer using different amounts of memory. C recognizes each of these data types, in addition to other types yet to be presented.
2. The `printf( )` function can be used to display all of C's data types. The conversion control sequences for displaying integer, floating point, double precision, and character values are `%d`, `%f`, `%lf`, and `%c`, respectively. The `%f` conversion control sequence can be used in place of the `%lf` sequence for double precision values.
3. Field width specifiers can be included with conversion control sequences to explicitly specify the format of displayed fields. This includes both the total width of the output field and, for floating point and double precision numbers, the number of decimal digits to display.
4. Every variable in a C program must be declared as to the type of value it can store. Declarations within a function must be placed as the first statements after a left brace, `{`. Variables may also be initialized when they are declared. Additionally, variables of the same type may be declared using a single declaration statement.
5. Declaration statements always play the software role of informing the compiler of a function's valid variable names. When a variable declaration also causes the compiler to set aside memory storage locations for the variable, the declaration statement is also called a *definition statement*. All of the declarations we have encountered have also been definition statements.
6. A simple C program containing declaration statements has the form:

```
#include <stdio.h>
main( )
{
  declaration statements;

  other statements;
}
```

7. An expression is any combination of constants and/or variables that can be evaluated to yield a value.
8. Expressions are evaluated according to the precedence and associativity of the operators used in the expression.
9. The assignment symbol, =, is an operator. Expressions using this operator assign a value to a variable; additionally, the expression itself takes on a value. Since assignment is an operation in C, multiple uses of the assignment operator are possible in the same expression.
10. The increment operator, ++, adds one to a variable, while the decrement operator, --, subtracts one from a variable. Both of these operators can be used as prefixes or postfixes. In prefix operation the variable is incremented (or decremented) before its value is used. In postfix operation the variable is incremented (or decremented) after its value is used.
11. Although assignment and output statements can be placed in any order within a program after the declaration statements, it only makes sense to use a `printf( )` function call to display the contents of a variable after a value has been assigned to it.

2.9 Enrichment Study: Variable Addresses and Memory Storage

Two major items associated with every variable are the value stored in it and the variable's address. In C, the value stored in a variable is formally referred to as the variable's *rvalue*, and the address of the variable is called the variable's *lvalue*. These two terms make more sense when a typical illustration of a variable is considered, as illustrated in Figure 2–20. As shown in this figure, the address of the variable is typically written on the left of the figure and the variable's contents (the value in the box) on the right.

Programmers are usually concerned only with the value assigned to a variable (its contents, or rvalue) and give little attention to where the value is stored (its address, or lvalue). For example, consider Program 2-14.

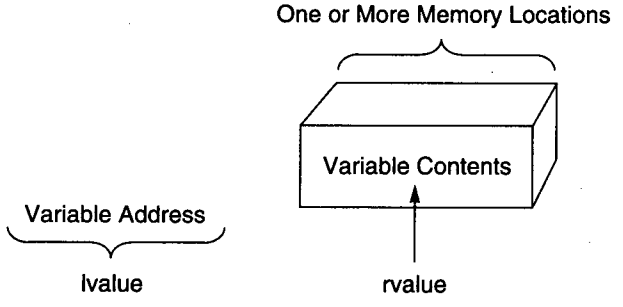


FIGURE 2–20 A Typical Variable

Program 2-14

```
#include <stdio.h>
main( )
{
  int num;

  num = 22;
  printf("The value stored in num is %d.",num);

}
```

All that Program 2-14 does is print out the value 22, which was assigned to the integer variable num. We can go further, however, and ask "Where is the number 22 actually stored?" Although the answer is "In num," this is only half of the answer. The variable name num is simply a convenient symbol for real, physical locations in memory, as illustrated in Figure 2–21.

To determine the address of num (its lvalue), we can use C's address operator, &, which means "the address of," directly in front of the variable name (no space between & and the variable). For example, &num means the address of num, &total means the address of total, and &price means the address of price. Program 2-15 uses the address operator to display the address of the variable num.

Program 2-15

```
#include <stdio.h>
main( )
{
  int num;

  num = 22;
  printf("num = %d  The address of num = %p.", num, &num);
}
```

The output of Program 2-15 is:

```
num = 22  The address of num = FFF0
```

Figure 2–22 illustrates the additional address information provided by the output of Program 2-15.

Clearly, the address output by Program 2-15 depends on the computer used to run the program. Every time Program 2-15 is executed, however, it displays a

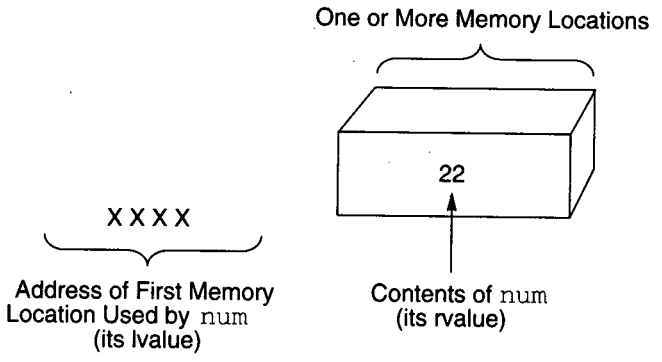
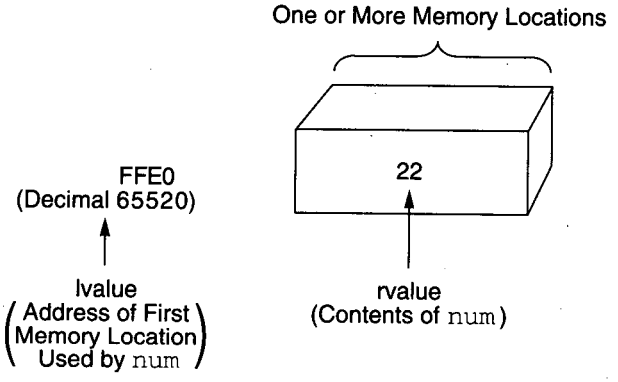


FIGURE 2–21 Somewhere in Memory

the hexadecimal address of the first byte used to store the variable num. Note also that the address is printed using the conversion control sequence `%p`. The `%p` conversion control sequence simply provides us with a convenient way of displaying an address. The display has no impact on how addresses are used internal to the program; it merely provides us with a useful representation that is helpful in understanding what addresses are.[6]

As we shall see, using addresses as opposed to only displaying them provides the C programmer with an extremely powerful programming tool. Addresses provide the ability to penetrate directly into the inner workings of the computer and access the computer's basic storage structure. This ability, which is more

FIGURE 2–22 A More Complete Picture of the Variable num



[6] In non-ANSI C, the `%p` conversion control sequence may not be available. For these compilers use the conversion sequence `%u`, which forces the address to be treated as an unsigned integer data type, and what is displayed is `printf( )`'s representation of the address in this format. An address, however, is not an unsigned integer data type—it is a unique data type that may or may not require the same amount of storage as an unsigned integer. The use of the `%u` conversion control sequence simply provides us with a convenient way of displaying an address when the `%p` is unavailable.

important to systems programmers than applications programmers, gives C programmers capabilities and programming power that are just not available in most other computer languages.

Determining Storage Size

In addition to allowing us to see a variable's address, C also provides an operator for determining the amount of storage allocated for each data type. This operator, called the `sizeof( )` operator, returns the number of bytes of the variable or data type included in the parentheses. Examples of the `sizeof( )` operator are:

```
sizeof(num1)   sizeof(int)   sizeof(float)
```

If the item in parentheses is a variable, as in the example `sizeof(num1)`, `sizeof( )` returns the number of bytes of storage that the computer reserved for the variable. If the item following the word `sizeof` is a data type, such as `int` or `float`, `sizeof( )` returns the number of bytes of storage that the computer uses for the given data type. Using either approach, we can use `sizeof` to determine the amount of storage used by different data types. Consider Program 2-16.

Program 2-16

```
#include <stdio.h>
main( )
{
  int num1;

  printf("\nBytes of storage used by an integer: %d",sizeof(num1));
}
```

Program 2-16 declares that the variable `num1` is used to store an integer. The `sizeof( )` operator is then used to tell us how much room the computer actually set aside for the variable num1. The `sizeof( )` operator itself is used as an argument to the `printf( )` function. When Program 2-16 is run on an IBM personal computer the following output is obtained:

```
Bytes of storage used by an integer: 2
```

In using the `sizeof( )` operator, one caution should be noted. The number of bytes given by the `sizeof( )` operator is in terms of the storage reserved for one character. In ANSI C the specified storage allocation is one byte per character, so the value returned by `siteof( )` is a true byte count.

Completing the Basics

Chapter Three

3.1 Mathematical Library Functions
3.2 The `scanf( )` Function
3.3 `scanf( )` with Buffered Input
3.4 Symbolic Constants
3.5 Applications
3.6 Common Programming Errors
3.7 Chapter Summary
3.8 Enrichment Study: Program Life Cycle

In the previous two chapters we explored how results are displayed using C's `printf( )` function and how numerical data are stored and processed using variables and assignment statements. In this chapter we complete our introduction to C by presenting additional processing and input capabilities.

3.1 Mathematical Library Functions

As we have seen, assignment statements can be used to perform arithmetic computations. For example, the assignment statement

```
volts = current * resist;
```

multiplies the value in `current` times the value in `resist` and then assigns the resulting value to `volts`. Although addition, subtraction, multiplication, and division are easily accomplished using C's arithmetic operators, no such operators exist for raising a number to a power, finding the square root of a number, or determining trigonometric values. To facilitate the calculation of powers, square roots, trigonometric, logarithmic, and other mathematical calculations frequently required in scientific and engineering programs, C provides standard preprogrammed functions that can be included in a program. Like the `printf( )` function with which you are already familiar, the available mathematical functions are stored in a system library that contains the collection of standard and tested functions available on your system.

Before using one of C's mathematical functions, you must know:

- The name of the desired mathematical function
- What the mathematical function does
- The type of data required by the mathematical function
- The data type of the result returned by the mathematical function

To illustrate the use of C's mathematical functions, consider the mathematical function named `sqrt`, which calculates the square root of a number. The square root of a number is computed using the expression

```
sqrt(number)
```

where the function's name, in this case `sqrt`, is followed by parentheses containing the number for which the square root is desired. The purpose of the parentheses following the function name is to provide a funnel through which data can be passed to the function (see Figure 3–1). The items that are passed to the function through the parentheses are called *arguments* of the function and constitute its input data. For example, the following expressions are used to compute the square root of the arguments `4.0`, `17.0`, `25.0`, `1043.29`, and `6.4516`:

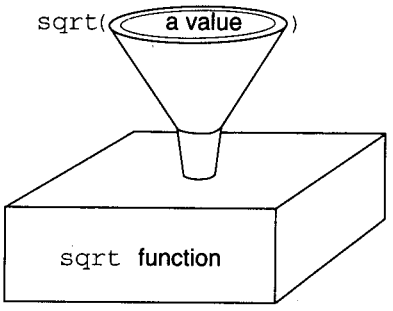


FIGURE 3–1 Passing Data to the `sqrt( )` Function

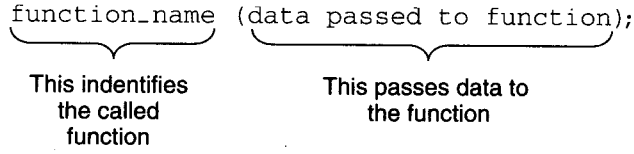
```
sqrt(4.0)
sqrt(17.0)
sqrt(25.0)
sqrt(1043.29)
sqrt(6.4516)
```

The argument to the function named `sqrt` must be a double precision value. The `sqrt` function computes the square root of its argument and the returned result is itself a double precision value. The values returned by the previous expressions are:

Expression	Value Returned
sqrt(4.0)	2.000000
sqrt(17.0)	4.123106
sqrt(25.0)	5.000000
sqrt(1043.29)	32.300000
sqrt(6.4516)	2.540000

In addition to the `sqrt` function, the more commonly used mathematical functions provided in C are listed in Table 3–1. Although some of the mathematical functions listed require more than one argument, all mathmatical functions, return a single value. Table 3–2 lists the value returned by selected functions using example arguments. Note that the argument types for the examples agree with those given in Table 3–1 for the specified function.

FIGURE 3–2 Using and Passing Data to a Function



When a mathematical function is used it is called into action by giving the name of the function and passing any data to it within the parentheses following the function's name (see Figure 3–2).

The arguments that are passed to a function need not be single constants. Expressions can also be arguments provided that the expression can be computed

TABLE 3–1 Commonly used mathmatical C Functions

Function name and argument(s)	Argument type(s)	Returned value	Description
`abs(i)`	integer	integer	Absolute value of `i`
`fabs(d)`	double	double	Absolute value of `d`
`pow(d1,d2)`	double	double	`d1` raised to the `d2` power
`exp(d)`	double	double	`e` raised to the `d` power
`sqrt(d)`	double	double	Square root of `d`
`sin(d)`	double	double	Sine of `d` (`d` in radians)
`cos(d)`	double	double	Cosine of `d` (`d` in radians)
`tan(d)`	double	double	Tangent of `d` (`d` in radians)
`tanh(d)`	double	double	Hyperbolic tangent of `d`
`log(d)`	double	double	Natural log of `d`
`log10(d)`	double	double	Common log (base 10) of d

TABLE 3–2 Selected Function Examples

Example	Returned value
`abs(-7.362)`	7.362000
`abs(-3)`	3
`pow(2.0,5.0)`	32.00000
`square(4.0)`	16.00000
`exp(-3.2)`	0.040762
`pow10(3)`	1000.000000
`log(18.697)`	2.928363
`log10(18.697)`	1.271772

to yield a value of the required data type. For example, the following arguments are valid for the given functions:

```
sqrt(4.0 + 5.3 * 4.0)     abs(2.3 * 4.6)
sqrt(16.0 * 2.0 - 6.7)    sin(theta - phi)
sqrt(x * y - z/3.2)       cos(2.0 * omega)
```

The expressions in parentheses are first evaluated to yield a specific value. Thus, values would have to be assigned to the variables `theta`, `phi`, `x`, `y`, `z`, and `omega` before their use in the above expressions. After the value of the argument is calculated, it is passed to the function.

Functions may be included as part of larger expressions. For example:

```
4 * sqrt(4.5 * 10.0 - 9.0) - 2.0 =
        4 * sqrt(36.000000) - 2.0 =
              4 * 6.000000 - 2.0 =
                 24.000000 - 2.0 = 22.000000
```

and

```
3.0 * log(30 * .514) =
    3.0 * log(15.42) =
      3.0 * 2.735665 = 8.206995
```

The step–by–step evaluation of

```
3.0 * sqrt(5 * 33 - 13.71) / 5
```

is:

Step	Result
1. Perform multiplication in argument	3.0 * sqrt(165 − 13.71) / 5
2. Complete argument calculation	3.0 * sqrt(151.290000) / 5
3. Return a function value	3.0 * 12.300000 / 5
4. Perform the multiplication	36.900000 / 5
5. Perform the division	7.380000

Program 3-1 illustrates the use of the `sqrt` function to determine the time it takes a ball to hit the ground after it has been dropped from an 800 foot tower. The mathematical formula used to calculate the time in seconds that it takes to fall a given distance in feet is:

$$\text{time} = \text{sqrt}(2 * \text{distance} / g)$$

where g is the gravitational constant equal to 32.2 ft/sec^2.

Program 3-1

```
#include <stdio.h>
main( )
{
  double time, height;
  double sqrt( );

  height = 800.0;
  time = sqrt(2.0 * height / 32.2);
  printf("\nIt will take %4.2lf seconds", time);
  printf("\nto fall %7.3lf feet.", height);
}
```

Notice that Program 3-1 contains a declaration statement for the `sqrt( )` function. Although the actual code for the `sqrt( )` function is contained in the standard system library, the proper declaration for the function must be provided by the programmer in any program using the function. Alternatively, all compilers have a standard header file named `math.h` that contains appropriate declaration statements for the supplied mathematical functions. To include the information in this file in your program, allowing you to use all the mathematical functions without explicitly typing declaration statements for each function used, the following preprocessor statement must be included with your program:

```
#include <math.h>
```
◄——— no semicolon

Thus, using this statement Program 3-1 can be rewritten as Program 3-2 (the order of the two `#include` commands can be reversed).

Program 3-2

```
#include <math.h>
#include <stdio.h>
main( )
{
  double time, height;

  height = 800.0;
  time = sqrt(2.0 * height / 32.2);
  printf("\nIt will take %4.2lf seconds", time);
  printf("\nto fall %7.3lf feet.", height);
}
```

Notice in Program 3-2 that the declaration statement `double sqrt( );` used in Program 3-1 has been eliminated. The output of both Programs 3-1 and 3-2 is:

```
It will take  7.05 seconds
to fall 800.00 feet.
```

As used in both programs, the value returned by the `sqrt( )` function is assigned to the variable `time`. In addition to assigning a function's returned value to a variable, the returned value may be included within a larger expression, or even used as an argument to another function. For example, the expression

```
sqrt (sin (abs (theta) ) )
```

is valid. Since parentheses are present the computation proceeds from the inner to the outer pairs of parentheses. Thus, the absolute value of `theta` is computed first and used as an argument to the `sin` function. The value returned by the `sin` function is then used as an argument to the `sqrt` function.

Note that the arguments of all trigonometric functions (`sin, cos`, etc.) must be in radians. Thus, to obtain the sine of an angle that is given in degrees the angle must first be converted to radian measure. This is easily accomplished by multiplying the angle by the term (3.1416/180.). For example, to obtain the sine of 30 degrees, the expression `sin( 30 * 3.1416/180.)` may be used.

Type Conversions

We have already seen the conversion of an operand's data type within mixed arithmetic expressions (see Section 2.1). For example, if `val` is a double precision variable and `num` is an integer variable, `num`'s value is converted to double precision in the expression `val + num`.

The general rules for converting operands in mixed arithmetic expressions were presented in the previous chapter. A more complete set of conversion rules for arithmetic operators is listed in Table 3–3.

TABLE 3–3 Conversion Rules for Arithmetic Operators

Rule 1.	All character and short integer operands are converted to integer values and all floating point operands are converted to double precision values in arithmetic expressions.
Rule 2.	If one operand is a double precision value, then the other operand is converted to a double precision value and the result of the expression is a double precision value.
Rule 3.	If one operand is a long integer value, then the other operand is converted to a long integer value and the resulting value of the expression is a long integer value.
Rule 4.	If one operand is an unsigned integer value, then the other operand is converted to an unsigned integer value and the resulting value of the expression is an unsigned value.
Rule 5.	If both operands are of type `int`, no conversions occur and the resulting value of the expression is an integer value.

Data type conversions also take place across assignment operators. Here the value of the expression on the right side of the equal sign is converted to the data type of the variable to the left of the equal sign. For example, consider the evaluation of the expression

```
a = b * d
```

where `a` and `b` are integer variables and `d` is a floating point variable. Referring to Rule 1 in Table 3–3, the value of `d` used in the expression is converted to a double precision number for purposes of computation (it is important to note that the value stored in `d` remains a floating point number). Since one of the operands is a double precision variable, Rule 2 provides that `b`'s value is converted to a double precision number for the computation (again, the value stored in `b` remains an integer) and the resulting value of the expression `b * d` is a double precision number. Finally, since the left side of the assignment operator is an integer variable, the double precision value of the expression `(b * d)` is truncated to an integer value and stored in the variable `a`.

In addition to data type conversions that are made automatically to operands in mixed arithmetic expressions, C also provides for user–specified type conversions. The operator used to force the conversion of a value to another type is the cast operator. This is a unary operator having the form `(data_type)`, where `data type` is the desired `data type` of the operand following the cast. For example, the expression

```
(int) (a * b)
```

ensures that the value of the expression `a * b` is converted to an integer value. The parentheses around the expression `(a * b)` are required because the cast operator has a higher precedence than the multiplication operator.

As a last example, consider the expression `(int) a * b`, where both `a` and `b` are double precision variables. Here, only `a`'s value is cast into an integer before multiplication by `b`. The cast into an integer value causes the fractional part of `a`'s value to be truncated. Since `b` is a double precision operand, the value of the operand `(int) a` is converted back to a double precision number (Rule 2 in Table 3–3). The forced conversion back to a double precision number, however, does not restore the fractional part of `a`. As before, the value stored in `a` is not affected and remains a double precision number; it is just the value of `a` used to evaluate the expression that is truncated.

Exercises 3.1

1. Write function calls to determine:
   a. The square root of 6.37.
   b. The square root of `x - y`.
   c. The sine of 30 degrees.
   d. The sine of 60 degrees.
   e. The absolute value of $a^2 - b^2$.
   f. The value of *e* raised to the 3rd power.

2. For $a = 10.6$, $b = 13.9$, $c = -3.42$, determine the value of:

a `(int) a`
b. `(int) b`
c. `(int) c`
d. `(int) a + b`
e. `(int) a + b + c`
f. `(int) (a + b) + c`
g. `(int) (a + b + c)`
h. `(float) (int) a + b`
i. `(float) (int) (a + b)`
j. `abs(a) + abs(B)`
k. `sqrt(abs(a - B))`

3. Write C statements for the following:

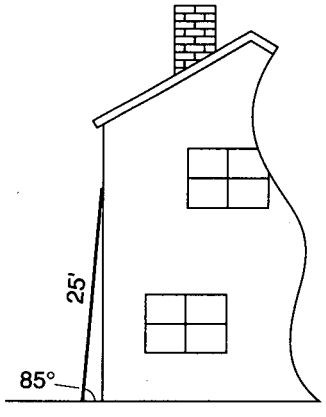
a. $b = \sin x - \cos x$
b. $b = \sin^2 x - \cos^2 x$
c. area = $(c * b * \sin a)/2$
d. $c = \sqrt{a^2 + b^2}$
e. $p = \sqrt{|m - n|}$
f. $\text{sum} = \dfrac{a(r^n - 1)}{r - 1}$

4. Write, compile, and execute a C program that calculates and returns the 4th root of the number 81.0, which is 3. When you have verified that your program works correctly, use it to determine the fourth root of 1,728.896400. Your program should make use of the `sqrt( )` function.

5. Write, compile, and execute a C program that calculates the distance between two points whose coordinates are (7,12) and (3,9). Use the fact that the distance between two points having coordinates $(x1,y1)$ and $(x2,y2)$ is distance = $\text{sqrt}([x1 - x2]^2 + [y1 - y2]^2)$. When you have verified that your program works correctly by manually calculating the distance between the two points, use your program to determine the distance between the points (-12,-15) and (22,5).

6. If a 20-foot ladder is placed on the side of a building at an 85 degree angle, as illustrated in Figure 3–3, the height at which the ladder touches the building can be calculated as height = 20 * sin 85°. Calculate this height by hand and then write, compile, and execute a C program that determines and displays the value of the height. When you have verified that your program works correctly, use it to determine the height of a 25-foot ladder placed at an angle of 85 degrees.

FIGURE 3–3 Calculating the Height of a Ladder Against a Building



7. The maximum height reached by a ball thrown with an initial velocity v in feet/sec at an angle of θ is given by the formula height = $(.5 * v^2 * \sin^2 \theta) / 32.2$. Using this formula, write, compile, and execute a C program that determines and displays the maximum height reached when the ball is thrown at 5 miles/hour at an angle of 60 degrees. (*Hint:* Make sure to convert the initial velocity into the correct units.) Calculate the maximum height manually and verify the result produced by your program. After you have verified that your program works correctly, use it to determine the height reached by a ball thrown at 7 miles/hour at an angle of 45 degrees.

8. For small values of x, the value of $\sin(x)$ can be approximated by the first three terms of a power series as:

$$x - \frac{x^3}{6} + \frac{x^5}{120}$$

As with the sin function, the value of x must be in radians. Using this power series, write, compile, and execute a C program that approximates the sine of 180/3.1416 degrees, which equals one radian. Additionally, have your program use the `sin` function to calculate the sine and display both calculated values and the absolute difference of the two results. Manually verify the approximation produced by your program. After you have verified your program is working correctly, use it to approximate the value of the sine of 62.2 degrees.

9. The polar coordinates of a point consist of the distance, r, from a specified origin and an angle, θ, with respect to the X axis. The x and y coordinates of the point are related to its polar coordinates by the formulas

$$x = r \cos \theta$$

$$y = r \sin \theta$$

Using these formulas, write a C program that calculates the x and y coordinates of the point whose polar coordinates are $r = 10$ and $\theta = 30$ degrees. Verify the results produced by your program by calculating the results manually. After you have verified your program is working correctly, use it to convert the polar coordinates $r = 12.5$ and $\theta = 67.8°$ into rectangular coordinates.

10. A model of worldwide population in billions of people after 1990 is given by the equation

$$\text{Population} = 5.5\, e^{.02\,[\text{Year} - 1990]}$$

Using this formula, write, compile, and execute a C program to estimate worldwide population in the year 1995. Verify the result displayed by your program by calculating the answer manually. After you have verified your program is working correctly, use it to estimate world population in the year 2012.

11. A model to estimate the number of grams of a certain radioactive isotope left after N years is given by the formula

$$\text{Remaining material} = (\text{Original material})\, e^{-.00012\,N}$$

Using this formula, write, compile, and execute a C program to determine the amount of radioactive material remaining after 1000 years, assuming an initial amount of 100 grams. Verify the display produced by your program using a hand calculation. After you have verified your program is working correctly, use it to determine the amount of radioactive material remaining after 275 years, assuming an initial amount of 250 grams.

12. The number of years that it takes for a certain isotope of uranium to decay to one–half of an original amount is given by the formula

$$\text{Half-life} = \ln(2)/k$$

where k equals .00012. Using this formula, write, compile, and execute a C program that calculates and displays the half-life of this uranium isotope. Verify the result produced by your program using a hand calculation. After you have verified your program is working correctly, use it to determine the half-life of a uranium isotope having $k = .00026$.

13. The amplification of electronic circuits is measured in units of decibels, which is calculated as

$$10 \log (PO/PI)$$

where *PO* is the power of the output signal and *PI* is the power of the input signal. Using this formula, write, compile, and execute a C program that calculates and displays the decibel amplification in which the output power is 50 times the input power. Verify the result displayed by your program using a hand calculation. After you have verified your program is working correctly, use it to determine the amplification of a circuit whose output power is 4.639 times its input power.

14. The loudness of a sound is measured in units of decibels, which is calculated as

$$10 \log (SL/RL)$$

where *SL* is intensity of the sound being measured and *RL* is a reference sound intensity level. Using this formula write a C program that calculates and displays the decibel loudness of a busy street having a sound intensity of 10,000,000 *RL*. Verify the result produced by your program using a hand calculation. After you have verified your program is working correctly, use it to determine the sound level in decibels of the following sounds:

a. A whisper of sound at intensity 200 *RL*
b. A rock band playing at a sound intensity of 1,000,000,000,000 *RL*
c. An airplane taking off at a sound intensity of 100,000,000,000,000 *RL*

15. The dollar change remaining after an amount `paid` is used to pay a restaurant check of amount `check` can be calculated using the following C statements:

```
/* determine the amount of pennies in the change */
        change = (paid - check) * 100;
/* determine the number of dollars in the change */
        dollars = (int) (change/100);
```

a. Using the previous statements as a starting point, write a C program that calculates the number of dollar bills, quarters, dimes, nickels, and pennies in the change when $10 is used to pay a bill of $6.07.
b. Without compiling or executing your program check the effect, by hand, of each statement in the program and determine what is stored in each variable as each statement is encountered.
c. When you have verified that your algorithm works correctly, compile and execute your program. Verify that the result produced by your program is correct. After you have verified your program is working correctly, use it to determine the change when a check of $12.36 is paid using a twenty-dollar bill.

16a. For display purposes the `%f` conversion control sequence allows the programmer to round all outputs to the desired number of decimal places. This can, however, yield seemingly incorrect results when used in financial program that require all monetary values to be displayed to the nearest penny. For example, the display produced by the statements

```
float a, b;
a = 1.674
b = 1.322
printf("\n%4.2f",a);
printf("\n%4.2f",b);
printf("\n——");
c = a + b;
printf("\n%4.2f",c);
```

is:

```
1.67
1.32
3.00
```

Clearly, the sum of the displayed numbers should be 2.99 and not 3.00. The problem is that although the values in a and b have been displayed with two decimal digits, they were added internal to the program as three–digit numbers. The solution is to round the values in a and b before they are added by the statement `c = a + b;`. Using the `(int)` cast, devise a method to round the values in the variables a and b to the nearest hundredth (penny value) before they are added.

b. Include the method you have devised for Exercise 16a into a working program that produces the following display:

```
1.67
1.32
2.99
```

3.2 The scanf ( ) Function

Data for programs that are only going to be executed once may be included directly in the program. For example, if we wanted to multiply the numbers 300.0 and .05, we could use Program 3-3.

Program 3-3

```
#include <stdio.h>
main( )
{
  float num1,num2,product;

  num1 = 300.0;
  num2 = .05;
  product = num1 * num2;
  printf("%f times %f is %f", num1, num2, product);
}
```

The output displayed by Program 3–3 is:

```
300.000000 times .050000 is 15.000000
```

Program 3-3 can be shortened, as illustrated in Program 3-4. Both programs, however, suffer from the same basic problem in that they must be rewritten in order to multiply different numbers. Neither program lets the user enter different numbers to be operated on.

Program 3-4

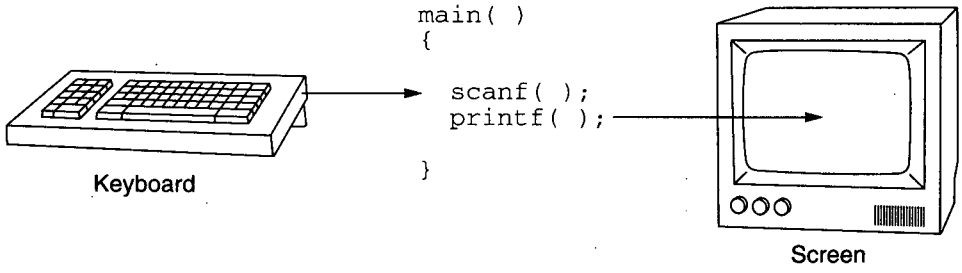
```
#include <stdio.h>
main( )
{
  printf("%f times %f is %f", 300.0, .05, 300.0*.05);
}
```

Except for the practice provided to the programmer of writing, entering, and running the program, programs that do the same calculation only once, on the same set of numbers, are clearly not very useful. After all, it is simpler to use a calculator to multiply two numbers than to enter and run either Program 3-3 or Program 3-4.

This section presents the `scanf( )` function, which is used to enter data into a program while it is executing. Just as the `printf( )` function displays a copy of the value stored inside a variable, the `scanf( )` function allows the user to enter a value at the terminal. The value is then stored directly in a variable.

Like the `printf( )` function, the `scanf( )` function requires a control string as the first argument inside the function name parentheses. The control string tells the function the type of data being input and uses the same conversion control sequences as the `printf( )` function. Unlike the control string used in a `printf( )` function, however, the control string passed to `scanf( )` typically consits of conversion control sequences only. Also, unlike `printf( )` where a list of variable names can follow the control string, `scanf( )` requires that a list

FIGURE 3–4 `scanf( )` Is Used to Enter Data; `printf( )` Is Used to Display Data



of variable addresses follow the control string. For the variables we have been using (integer, floating point, double, and character) the variable addresses are obtained by writing an ampersand symbol, &, immediately before the variable's name. For example, whereas num1, num2, num3 is a list of three variable names, &num1, &num2, &num3 is a list of three variable addresses. The ampersand symbol, &, is C's address operator, which means "the address of." Thus, if num1 is a variable name, &num1 means "the address of num1," and the statement scanf("%d", &num1); is a call to the scanf( ) function.[1] The conversion control sequence %d in this statement is identical to the conversion control sequence used in printf( ) in that it tells the scanf( ) function that it will be dealing with an integer number, and the address operator & in front of the variable num1, as already noted, is required for scanf( ).

When a statement such as scanf("%d",&num1); is encountered, the computer stops program execution and continuously scans the keyboard for data (scanf is short for scan function and formatted scan). When a data item is typed, the scanf( ) function stores the item using the address it was given. The program then continues execution with the next statement after the call to scanf( ). To see this, consider Program 3-5.

Program 3-5

```
#include <stdio.h>
main( )
{
  float num1, num2, product;

  printf("Please type in a number: ");
  scanf("%f",&num1);
  printf("Please type in another number: ");
  scanf("%f",&num2);
  product = num1 * num2;
  printf("%f times %f is %f",num1, num2,product);
```

The first call to printf( ) in Program 3-5 prints a message that tells the person at the terminal what should be typed. When a message is used in this manner it is called a *prompt*. In this case the prompt tells the user to type a number. The computer then executes the next statement, which is a call to scanf( ). The scanf( ) function puts the computer into a temporary pause (or wait) state for as long as it takes the user to type a value. Then the user signals the scanf( ) function by pressing the return key after the value has been typed. The entered

[1] The interested reader may review Section 2.9, which contains more detailed introductory material on the address operator. We will encounter the address operator again and be much more specific about its purpose in Chapters 6 and 10.

value is stored in the variable whose address was passed to `scanf( )`, and the computer is taken out of its paused state. Program execution proceeds with the next statement, which in Program 3-5 is another call to `printf( )`. This call causes the next message to be displayed. The second call to `scanf( )` again puts the computer into a temporary wait state while the user types a second value. This second number is stored in the variable `num2`.

The following sample run was made using Program 3-5.

```
Please type in a number: 300.
Please type in another number: .05
300.000000 times .050000 is 15.000000
```

In Program 3-5, each call to `scanf( )` is used to store one value in a variable. The `scanf( )` function, however, can be used to enter and store as many values as there are conversion control sequences in the control string. For example, the statement

```
scanf("%f %f",&num1,&num2);
```

results in two values being read from the terminal and assigned to the variables `num1` and `num2`. If the data entered at the terminal was

```
0.052 245.79
```

the variables `num1` and `num2` would contain the values `0.052` and `245.79`, respectively. The space in the control string between the two conversion control sequences, `"%f %f"`, is strictly for readability. The control string `"%f%f"` would work equally well. When actually entering numbers such as 0.052 and 245.79, however, you should leave at least one space between the numbers, regardless of which control string, `"%f %f"` or `"%f%f"`, is used. The space between the entered numbers, called a *delimiter*, clearly indicates where one number ends and the next begins. Inserting more than one space between numbers has the same effect on `scanf( )` as inserting a single space.

The only time that a space can affect the value being entered is when `scanf( )` is expecting a character data type. For example, the statement `scanf("%c%c%c",&ch1,&ch2,&ch3);` causes `scanf( )` to store the next three characters typed in the variables `ch1`, `ch2`, and `ch3`, respectively. If you type `x y z`, the `x` is stored in `ch1`, a blank is stored in `ch2`, and `y` is stored in `ch3`. If, however, the statement `scanf("%c %c %c",&ch1,&ch2,&ch3);` was used, `scanf( )` looks for three characters, each separated by exactly one space.

Any number of `scanf( )` function calls may be made in a program, and any number of values may be input using a single `scanf( )` function. Just be sure that a conversion control sequence is used for each value to be entered and that the address operator is used in front of the variable name where the value is to be stored. Program 3-6 illustrates using the `scanf( )` function to input three numbers from the keyboard. The program then calculates and displays the average of the numbers entered.

Program 3-6

```
#include <stdio.h>
main( )
{
  int num1, num2, num3;
  float average;

  printf("Enter three integer numbers: ");
  scanf("%d %d %d", &num1, &num2, &num3);
  average = (num1 + num2 + num3)/3.0;
  printf("The average of the numbers is %f", average);
}
```

The following sample run was made using Program 3-6:

```
Enter three integer numbers: 22 56 73
The average of the numbers is 50.333333
```

Note that the data typed at the keyboard for this sample run consists of this input:

```
22 56 73
```

In response to this line of input, Program 3-6 stores the value 22 in the variable num1, the value 56 in the variable num2, and the value 73 in the variable num3 (see Figure 3–5). Since the average of three integer numbers can be a floating point number, the variable average, which is used to store the average, is declared as a floating point variable. Note also that the parentheses are needed in the assignment statement average = (num1 + num2 + num3)/3.0;. Without these parentheses, the only value that would be divided by three would be the integer in num3 (since division has a higher precedence than addition).

The conversion control sequences used in a scanf( ) control string are the same as those used in printf( ) calls, with one caution. In printing a double precision number using printf( ), the conversion control sequence for a floating point variable, %f, can be used. This is not true when using scanf( ). If a double precision number is to be entered, the conversion control sequence %lf must be used.

The scanf( ) function, again like the printf( ) function, does not test the data type of the values being entered. It is up to the user to ensure that all variables are declared correctly and that any numbers entered are of the correct type. However, scanf( ) is "clever" enough to make a few data type conversions. For example, if an integer is entered in place of a floating point or double precision number, the scanf( ) function automatically adds a decimal point at the end of the integer before storing the number. Similarly, if a floating point or double pre-

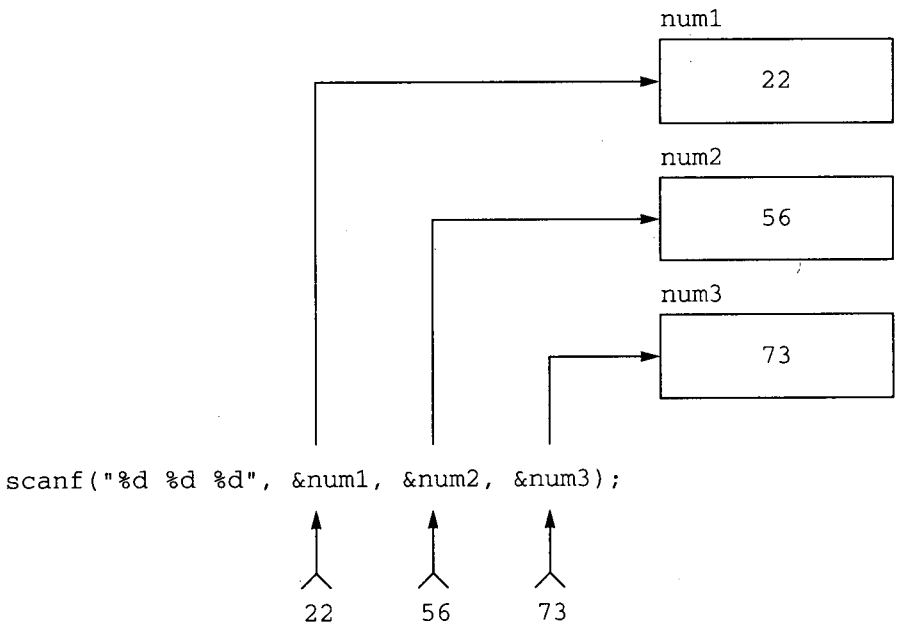


FIGURE 3–5 Inputting Data into the Variables num1, num2, and num3

cision number is entered when an integer is expected, the `scanf( )` function only uses the integer part of the number. For example, assume the following numbers are typed in response to the function call `scanf("%f %d %f", &num1,&num2,&num3);`:

```
56  22.879  33.923
```

`scanf( )` converts the `56` to `56.0` and stores this value in the variable `num1`. The function continues scanning the input, expecting an integer value. As far as `scanf( )` is concerned, the decimal point after the `22` in the number `22.879` indicates the end of an integer and the start of a decimal number. Thus, the number `22` is stored in `num2`. Continuing to scan the typed input, `scanf( )` takes the `.879` as the next floating point number and stores this value in `num3`. As far as `scanf( )` is concerned, `33.923` is extra input and is ignored. If, though, you do not initially type enough data, the `scanf( )` function will continue to make the computer pause until sufficient data has been entered.

Exercises 3.2

1. For the following declaration statements, write a `scanf( )` function call that will cause the computer to pause while the appropriate data is typed by the user.

a. `int firstnum;`

b. `float grade;`
c. `double secnum;`
d. `char keyval;`
e.
```
int month years;
float average;
```
f.
```
char ch;
int num1,num2;
double grade1,grade2;
```
g.
```
float interest, principal, capital;
double price,yield;
```
h.
```
char ch,letter1,letter2;
int num1,num2,num3;
```
i.
```
float temp1,temp2,temp3;
double volts1,volts2;
```

2. For the following scanf( ) function calls, write appropriate declaration statements for the variables.

a. `scanf("%d",&day);`
b. `scanf("%c",&fir_char);`
c. `scanf("%f",&grade);`
d. `scanf("%lf",&price);`
e. `scanf("%d %d %c",&num1,&num2,&ch1);`
f. `scanf("%f %f %d",&firstnum,&secnum,&count);`
g. `scanf("%c %c %d %lf",&ch1,&ch2,&flag,&average);`

3. Given the following declaration statements,

```
int num1, num2;
float firstnum, secnum;
double price, yield;
```

determine and correct the errors in the following `scanf( )` function calls.

a. `scanf("%d",num1);`
b. `scanf("%f %f %f",&num1,firstnum,&price);`
c. `scanf("%c %lf %f",&num1,&secnum,&price);`
d. `scanf("%d %d %lf",num1,num2,yield);`
e. `scanf(&num1,&num2);`
f. `scanf(&num1,"%d");`

4a. Write a C program that first displays the following prompt:

```
Enter the temperature in degrees Celsius:
```

Have your program accept a value entered from the keyboard and convert the temperature entered to degrees Fahrenheit, using the equation *Fahrenheit = (9.0 / 5.0) * Celsius + 32.0*. Your program should then display the temperature in degrees Celsius, using an appropriate output message.

b. Compile and execute the program written for Exercise 4a. Verify your program by calculating, first manually and then using your program, the Fahrenheit equivalent of the following test data:

Test data set 1: 0 degrees Celsius.
Test data set 2: 50 degrees Celsius
Test data set 3: 100 degrees Celsius

When you are sure your program is working correctly, use it to complete the following table:

Celsius	Fahrenheit
45	
50	
55	
60	
65	
70	

5. Write, compile, and execute a C program that displays the following prompt:

```
Enter the radius of a circle:
```

After accepting a value for the radius, your program should calculate and display the area of the circle. (*Note: area = 3.1416 * radius*^{2}.) For testing purposes, verify your program using a test input radius of 3 inches. After manually checking that the result produced by your program is correct, use your program to complete the following table:

Radius (inches)	Area (square inches)
1.0	
1.5	
2.0	
2.5	
3.0	
3.5	

6a. Write, compile, and execute a C program that displays the following prompts:

```
Enter the miles driven:
Enter the gallons of gas used:
```

After each prompt is displayed, your program should use a `scanf( )` statement to accept data from the keyboard for the displayed prompt. After the gallons of gas used has been entered, your program should calculate and display miles per gallon (MPG) obtained. This value should be included in an appropriate message and calculated using the equation *miles per gallon = miles / gallons used.* Verify your program using the following test data:

Test data set 1: Miles = 276, Gas = 10 gallons.
Test data set 2: Miles = 200, Gas = 15.5 gallons.

When you have completed your verification, use your program to complete the following table:

Miles driven	Gallons used	MPG
250	16.00	
275	18.00	
312	19.54	
296	17.39	

b. For the program written for Exercise 6a, determine how many verification runs are required to ensure the program is working correctly and give a reason supporting your answer.

7a. Write, compile, and execute a C program that displays the following prompts:

```
Enter a number:
Enter a second number:
Enter a third number:
Enter a fourth number:
```

After each prompt is displayed, your program should use a `scanf( )` statement to accept a number from the keyboard for the displayed prompt. After the fourth number has been entered, your program should calculate and display the average of the numbers. The average should be included in an appropriate message. Check the average displayed by your program using the following test data:

Test data set 1: 100, 100, 100, 100
Test data set 2: 100, 0, 100, 0

When you have completed your verification, use your program to complete the following table:

Numbers	Average
92, 98, 79, 85	
86, 84, 75, 86	
63, 85, 74, 82	

b. Repeat Exercise 7a, making sure that you use the same variable name, `number`, for each number input. Also use the variable `sum` for the sum of the numbers. (*Hint:* To do this, you may use the statement `sum = sum + number` after each number is accepted. Review the material on accumulating presented in Section 2.3.)

8a. Write, compile, and execute a C program that computes and displays the value of the second-order polynomial $ax^2 + bx + c$ for any user input values of the coefficients a, b, c, and the variable x. Have your program first display a message informing the user what the program will do, and then display suitable prompts to alert the user to enter the desired data. (*Hint:* Use a prompt such as `Enter the coefficient of the x squared term:.`)

b. Check the result produced by your program using the following test data:

Test data set 1: $a = 0, b = 0, c = 22, x = 56$
Test data set 2: $a = 0, b = 22, c = 0, x = 2$
Test data set 3: $a = 22, b = 0, c = 0, x = 2$
Test data set 4: $a = 2, b = 4, c = 5, x = 2$
Test data set 5: $a = 5, b = -3, c = 2, x = 1$

When you have completed your verification, use your program to complete the following table:

a	*b*	*c*	*x*	**Polynomial Value**
2.0	17.0	−12.0	1.3	
3.2	2.0	15.0	2.5	
3.2	2.0	15.0	−2.5	
−2.0	10.0	0.0	2.0	
−2.0	10.0	0.0	4.0	
−2.0	10.0	0.0	5.0	
−2.0	10.0	0.0	6.0	
5.0	22.0	18.0	8.3	
4.2	−16	−20	−5.2	

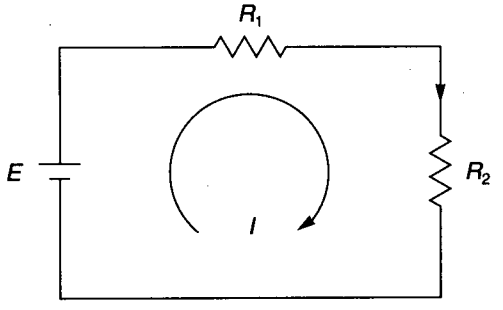
9. The number of bacteria, B, in a certain culture that is subject to refrigeration can be approximated by the equation $B = 300{,}000\, e^{-.032t}$, where e is the irrational number 2.71828 rounded to five decimal places, known as Euler's number, and t is the time in hours that the culture has been refrigerated. Using this equation, write, compile, and execute a single C program that prompts the user for a value of time, calculates the number of bacteria in the culture, and displays the result. For testing purposes, check your program using a test input of 10 hours. When you have verified the operation of your program, use it to determine the number of bacteria in the culture after 12, 18, 24, 36, 48, and 72 hours.

10. Write, compile, and execute a C program that calculates and displays the square root of a user-entered real number. Verify your program by calculating the square roots of the following data: 25, 16, 0, and 2. When you complete your verification, use your program to determine the square root of 32.25, 42, 48, 55, 63, and 79.

11. Write, compile, and execute a C program that calculates and displays the fourth root of a user-entered number. Recall from elementary algebra that the fourth root of a number can be found by raising the number to the 1/4 power. (*Hint:* Do not use integer division—can you see why?) Verify your program by calculating the fourth root of the following data: 81, 16, 1, and 0. When you have completed your verification, use your program to determine the fourth root of 42, 121, 256, 587, 1240, and 16256.

12. For the series circuit shown in Figure 3–6, the voltage drop, $V2$, across resistor $R2$ and the power, $P2$, delivered to this resistor are given by the equations $V2 = I\,R2$ and $P2 = I$

FIGURE 3–6 Calculating the Voltage Drop



V2, where $I = E/(R1 + R2)$. Using these equations, write, compile, and execute a C program that prompts the user for values of *E*, *R1*, and *R2*, calculates the voltage drop and power delivered to *R2*, and displays the results. Check your program using this test data: $E = 10$ volts, $R1 = 100$ ohms, and $R2 = 200$ ohms. When you have completed your verification, use your program to complete the following table:

R1 (volts)	R2 (ohms)	R3 (ohms)	Combined Resistance (ohms)
3000	3000	3000	
6000	6000	6000	
2000	3000	1000	
2000	4000	5000	
4000	2000	1000	
10000	100	100	

13. Write, compile, and execute a C program that computes the combined resistance of three parallel resistors. The values of each resistor should be accepted using a `scanf( )` statement (use the formula for combined resistance given in Exercise 24 of Section 2.3). Verify the operation of your program by using the following test data:

Test data set 1: $R1 = 1000$, $R2 = 1000$, and $R3 = 1000$.
Test data set 2: $R1 = 1000$, $R2 = 1500$, and $R3 = 500$.

When you have completed your verification, use your program to complete the following table:

E (volts)	*R1* (ohms)	*R2* (ohms)	Voltage Drop (volts)	Power Delivered (watts)
10	100	100		
10	100	200		
10	200	200		
20	100	100		
20	100	200		
20	200	200		

14. Using `scanf( )` statements, write, compile, and execute a C program that accepts the *x* and *y* coordinates of two points. Have your program determine and display the midpoints of the two points (use the formula given in Exercise 26 of Section 2.3). Verify your program using the following test data:

Test data set 1: Point 1 = (0,0) and Point 2 = (16,0)
Test data set 2: Point 1 = (0,0) and Point 2 = (0,16)
Test data set 3: Point 1 = (0,0) and Point 2 = (-16,0)
Test data set 4: Point 1 = (0,0) and Point 2 = (0,-16)
Test data set 5: Point 1 = (-5,-5) and Point 2 = (5,5)

When you have completed your verification, use your program to complete the following table.

Point 1	Point 2	Midpoint
(4,6)	(16,18)	
(22,3)	(8,12)	
(−10,8)	(14,4)	
(−12,2)	(14,−4)	
(−4,−6)	(20,16)	
(−4,−6)	(−16,−18)	

15. Write, compile, and execute a C program that calculates and displays the amount of money, A, available in N years when an initial deposit of X dollars is deposited in a bank account paying an annual interest rate of R percent. Use the relationship $A = X(1.0 + R/100)^N$. The program should prompt the user to enter appropriate values and use `scanf( )` statements to accept the data. In constructing your prompts use statements such as `Enter the amount of the initial deposit.` Verify the operation of your program by calculating, by hand and using your program, the amount of money available for the following test cases:

Test data set 1: $1000 invested for 10 years at 0% interest
Test data set 2: $1000 invested for 10 years at 6% interest

When you have completed your verification, use your program to determine the amount of money available for the following cases:

a. $1000 invested for 10 years at 8% interest.
b. $1000 invested for 10 years at 10% interest.
c. $1000 invested for 10 years at 12% interest.
d. $5000 invested for 15 years at 8% interest.
e. $5000 invested for 15 years at 10% interest.
f. $5000 invested for 15 years at 12% interest.
g. $24 invested for 300 years at 4% interest.

16. Program 3-5 prompts the user to input two numbers, where the first value entered is stored in `num1` and the second value is stored in `num2`. Using this program as a starting point, write a program that swaps the values stored in the two variables. (Hint: For an appropriate algorithm review the solution to Exercise 3 in Section 1.1)

17. Write a C program that prompts the user to type in an integer number. Have your program accept the number as an integer and immediately display the integer. Run your program three times. The first time you run the program enter a valid integer number, the second time enter a floating point number, and the third time enter a character constant. Using the output display, see what number your program actually accepted from the data you entered.

18. Repeat Exercise 17 but have your program declare the variable used to store the number as a floating point variable. Run the program four times. The first time enter an integer, the second time enter a decimal number with less than six decimal places, the third time enter a number with more than six decimal places, and the fourth time enter a character constant. Using the output display, keep track of what number your program actually accepted from the data you typed in. What happened, if anything, and why?

19. Write a C program that uses the declaration statement `double num;`. Then use the function call `scanf("%f",&num);` to input a value into `num`. (Notice that we have used the wrong conversion sequence for the variable `num`.) Run your program and enter a decimal number. Using a `printf( )` function call, have your program display the number stored in `num1`. Determine what problem you can run into when an incorrect conversion sequence is used in `scanf( )`.

20a. Why do you think that most successful applications programs contain extensive data input validity checks? (*Hint:* Review Exercises 17, 18, and 19.)

b. What do you think is the difference between a data type check and a data reasonableness check?
c. Assume that a program requests that the velocity and acceleration of a car be entered by the user. What are some checks that could be made on the data entered?

3.3 scanf( ) with Buffered Input[2]

Seemingly strange results are sometimes obtained when the `scanf( )` function is used to accept characters. To see how this can occur, consider Program 3-7, which uses `scanf( )` to accept the next character entered at the keyboard and store the character in the variable `fkey`.

Program 3-7

```
#include <stdio.h>
main( )
{
  char fkey;

  printf("Type in a character: ");
  scanf("%c", &fkey);
  printf("The key just accepted is %d", fkey);
}
```

When Program 3-7 is run, the character entered in response to the prompt `Type in a character:` is stored in the character variable `fkey` and the decimal code for the character is displayed by the last `printf( )` function call. The following sample run illustrates this:

```
Type in a character: m
The key just accepted is 109
```

At this point, everything seems to be working just fine, although you might be wondering why we displayed the decimal value of `m` rather than the character itself. The reason for this will soon become apparent.

When you type `m`, you usually press two keys, the m key and the ENTER key. On most computer systems these two characters are stored in a temporary holding area called a buffer immediately after you press the keys, as illustrated in Figure 3–7.

[2] This section contains supplementary material on the `scanf( )` function and can be omitted on first reading without loss of subject continuity.

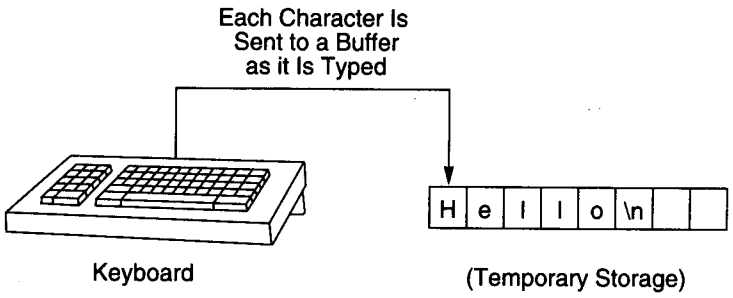


FIGURE 3–7 Typed Keyboard Characters are First Stored in a Buffer

The first key pressed, m in this case, is taken from the buffer and stored in `fkey`. This, however, still leaves the code for the ENTER key in the buffer. Any subsequent call to `scanf( )` for a character input automatically picks up the code for the ENTER key as the next character. For example, consider Program 3-8.

Program 3-8

```
#include <stdio.h>
main( )
{
  char fkey, skey;

  printf("Type in a character: ");
  scanf("%c", &fkey);
  printf("The key just accepted is %d", fkey);
  printf("\nType in another character: ");
  scanf("%c", &skey);
  printf("The key just accepted is %d", skey);
}
```

The following is a sample run for Program 3-8:

```
Type in a character: m
The key just accepted is 109
Type in another character: The key just accepted is 10
```

Let us review what has happened. When you enter `m` in response to the first prompt, you also press the ENTER key. From a character standpoint this represents the entry of two distinct characters. The first character is `m`, which is stored as `109`. The second character also gets stored in the buffer with the numerical code for the ENTER key. The second call to `scanf( )` picks up this code immediately, without waiting for any additional key to be pressed. The last call to

printf( ) displays the code for this key. The reason for displaying the numerical code rather than the character itself is because the ENTER key has no printable character associated with it that can be displayed.

Remember that every key has a numerical code, including the ENTER, SPACE, ESCAPE, and CONTROL keys. These keys generally have no effect when entering numbers, because scanf( ) ignores them as leading or trailing input with numerical data. Nor do these keys affect the entry of a single character requested as the first user data to be input, as is the case in Program 3-7. Only when a character is requested after the user has already input some other data, as in Program 3-8, does the usually invisible ENTER key become noticeable.

There is a quick solution to avoid having the ENTER key accepted as a legitimate character input. All we have to do is accept the ENTER key, store it as a character variable, and then just not use it. Program 3-9 illustrates this technique. The ENTER key is accepted along with the first character typed. This clears the computer's buffer and prepares the way for the character input.

Program 3-9

```
#include <stdio.h>
main( )
{
  char fkey, skey;

  printf("Type in a character: ");
  scanf("%c%c", &fkey, &skey);     /* the enter code goes to skey */
  printf("The key just accepted is %d", fkey);
  printf("\nType in another character: ");
  scanf("%c", &skey);   /* accept another code */
  printf("The key just accepted is %d", skey);
}
```

In reviewing Program 3-9, observe that the first scanf( ) function call accepts two back-to-back characters. Now when the user types m and presses the ENTER key, the m is assigned to fkey and the code for the ENTER key is automatically assigned to skey. The next call to scanf( ) stores the code for the next key pressed in the variable skey. also. This automatically erases the code for the ENTER key that was previously stored there. From the user's standpoint, the ENTER key has no effect except to signal the end of each character input. The following is a sample run for Program 3-9:

```
Type in a character: m
The key just accepted is 109
Type in another character: b
The key just accepted is 98
```

The solution to the "phantom" ENTER key used in Program 3-9 is not the only

solution possible (there is never just one way of doing something in C).[3] All solutions, however, center on the fact that the ENTER key is a legitimate character input and must be treated as such when using a buffered system.

3.4 Symbolic Constants

Literal data is any data within a program that explicitly identifies itself. For example, the constants `2` and `3.1416` in the assignment statement

```
circum = 2 * 3.1416 * radius;
```

are also called literals because they are literally included directly in the statement. Additional examples of literals are contained in the following C assignment statements. See if you can identify them.

```
perimeter = 2 * length * width;
        y = (5 * p) / 7.2;
 salestax = 0.05 * purchase;
```

The literals are the numbers `2`, `5` and `7.2`, and `0.05` in the first, second, and third statements, respectively.

Quite frequently, literal data used within a program have a more general meaning that is recognized outside the context of the program. Examples of these types of constants include the number 3.1416, which is the value of π accurate to four decimal places; 32.2 ft/sec^2, which is the gravitational constant (see, for example, Program 3-1); and the number 2.71828, which is Euler's number accurate to five decimal places.

The meaning of certain other constants appearing in a program are defined strictly within the context of the application being programmed. For example, in a program to determine bank interest charges, the value of the interest rate takes on a special meaning. Similarly, in determining the weight of various sized objects, the density of the material being used takes on a special significance. Numbers such as these are referred to by programmers as *magic numbers*. By themselves the numbers are quite ordinary, but in the context of a particular application they have a special ("magical") meaning. Frequently, the same magic number appears repeatedly within the same program. This recurrence of the same constant throughout a program is a potential source of error should the constant have to be changed. For example, if either the interest rate changes or a new material is employed with a different density, the programmer would have the cumbersome task of changing the value of the magic number everywhere it appears in the program. Multiple changes, however, are subject to error: If just one value is overlooked and not changed, the result obtained when the program is run will be incorrect and the source of the error troublesome to find.

To avoid the problems of having a magic number spread throughout a program and to permit clear identification of more universal constants, such as π, C allows the programmer to give these constants their own symbolic name. Then,

[3] Another solution, for example, is to replace the last `scanf( )` call in Program 3-9 with the statement `scanf("/n%c", &skey);`.

instead of using the constant throughout the program, the symbolic name is used instead. If the number ever has to be changed, the change need only be made once at the point where the symbolic name is equated to the actual constant value. Equating numbers to symbolic names is accomplished using a preprocessor `#define` statement. Two such statements are:

```
#define PI   3.1416      <-------- no semicolon
#define NUM_ELS  10      <-------- no semicolon
```

These two statements are called either `#define` or *equivalence* statements. Since they are preprocessor statements they must not be terminated with a semicolon. The first `#define` statement equates the value `3.1416` to the symbolic name `PI`, while the second `#define` statement equates the number `10` to the symbolic name `NUM_ELS`. Other terms for symbolic names are *named constants* or *symbolic constants.*

Although we have typed the symbolic constants in uppercase letters, lowercase letters could have been used. It is common in C, however, to use uppercase letters for symbolic constants. Then, whenever a programmer sees uppercase letters in a program, he or she knows the name is a symbolic constant defined in a `#define` statement, not a variable name declared in a declaration statement.

Once a constant has been named, the name can be used in any C statement in place of the number itself. For example, the assignment statement

```
circum = 2 * PI * radius;
```

makes use of the symbolic constant `PI`. This statement must, of course, appear after `PI` has been named in a `#define` statement. Usually `#define` statements are placed at the top of file before any functions, including `main( )`, are typed[4]. Program 3-10 illustrates the use of a `#define` statement to calculate the weight of a steel cylinder. The density of the steel is 0.284 lb/in^3.

Notice in Program 3-10 that each symbolic constant, `PI` and `DENSITY`, requires its own `#define` statement. The following run was made using Program 3-10 to determine the weight of a cylinder having a radius of 3 inches and a height of 12 inches.

```
Enter the radius of the cylinder: 3
Enter the height of the cylinder: 12

The cylinder weights 96.359154 pounds.
```

The advantage of using the symbolic constant `PI` in Program 3-10 is that it clearly identifies the value `3.1416` in terms recognizable to most people. The advantage of using the symbolic constant `DENSITY` is that it permits a programmer to change the value of the density for another material without having to search through the program to see where the density is used. If, of course, many different materials are to be considered, the density should be changed from a symbolic constant to a variable. A natural question arises, then, as to the difference between symbolic constants and variables.

[4] `#define` and `#include` statements may be freely intermixed in any order. Thus, in Program 3-10, the `#include` statement could have been placed above the `#define` statement.

Program 3-10

```
#define PI 3.1416
#define DENSITY 0.284
#include <stdio.h>
#include <math.h>
main( )
/* this program determines the weight of a steel cylinder */
/* by multiplying the volume of the cylinder times its density */
{
   float radius, height, weight;

   printf("Enter the radius of the cylinder: ");
   scanf("%f", &radius);
   printf("Enter the height of the cylinder: ");
   scanf("%f", &height);
   weight = PI * DENSITY * pow(radius,2.0) * height;
   printf("\nThe cylinder weighs %f pounds.", weight);
```

The value of a variable can be altered anywhere within a program. By its nature a symbolic constant is a constant value that must not be altered after it is defined. Naming a constant rather than assigning the value to a variable ensures that the value in the constant cannot be subsequently altered. Whenever a symbolic constant appears in an instruction it has the same effect as the constant it represents. Thus, `DENSITY` in Program 3-10 is simply another way of representing the number `0.284`. Since `DENSITY` and the number `0.284` are equivalent, the value of `DENSITY` may not be subsequently changed within the program. An assignment statement such as

```
DENSITY = 0.156;
```

is meaningless and will result in an error message, because `DENSITY` is not a variable. Since `DENSITY` is only a stand-in for the value `0.284`, this statement is equivalent to writing the invalid statement `0.284 = 0.156`.

Notice also that `#define` statements do not end with a semicolon. The reason for this is that `#define` statements are not processed by the regular C compiler used to translate C statements into machine language. The # sign is a signal to the C preprocessor. This preprocessor screens all program statements when a C program is compiled. When the preprocessor encounters a # sign, it recognizes an instruction to itself. The word `define` tells the preprocessor to equate the symbolic constant in the statement with the information or data following it. In the case of a statement like `#define PI 3.1416`, the word `PI` is equated to the value `3.1416`. The preprocessor then replaces each subsequent occurrence of the word `PI` in the C program with the value `3.1416`.

Notice that if a semicolon followed the literal value `3.1416`, the preprocessor would equate the word `PI` with `3.1416;`. Then, when it replaced `PI` in the assignment statement

```
weight = PI * DENSITY * pow(radius,2.0) * height;
```

the statement would become

```
weight = 3.1416; * DENSITY * pow(radius,2.0) * height;
```

which is the valid statement `weight = 3.1416;` followed by the invalid statement `* DENSITY * pow(radius,2.0) * height;`.

In addition to using a `#define` statement to name constants, as in Program 3-10, this statement can also be used to equate any expression or text to a symbolic name. For example, the statement

```
#define CONVERT 3.1416/180.0
```

equates the expression `3.1416/180.0` to the symbolic name `CONVERT`. As the expression `3.1416/180.0` is required for converting degrees to radians, the symbolic name selected for this conversion factor can be conveniently used whenever such a conversion is required. For example, in the assignment statement

```
height = distance * sin(CONVERT*angle);
```

the symbolic constant `CONVERT` is used to convert the value in `angle` to radian measure.

A previously defined symbolic constant can also be used in a subsequent `#define`. For example, the following sequence of preprocessor statements is valid:

```
#define PI 3.1416
#define CONVERT PI/180.0
```

Since the constant `3.1416` has been equated to the symbolic name `PI`, it can be used legitimately in any subsequent definition. Program 3-11 uses the symbolic

Program 3-11

```
#define PI 3.1416
#define CONVERT PI/180.0
#include stdio.h
#include math.h
main( )
{
  float angle;

  printf("Enter the angle (in degrees): ");
  scanf("%f", &angle);
  printf("\nThe sine of the angle is %f", sin(CONVERT*angle));
```

constant CONVERT to convert a user-entered angle in degrees into its equivalent radian measure for use by the sin( ) function.

Following is a sample run using Program 3-11:

```
Enter the angle (in degrees): 30
The sine of the angle is 0.500000
```

Realizing that #define statements simply relate two items allows us to use them to create individualized programming languages. For example, the #define statements

```
#define BEGIN {
#define END }
```

equate the first brace { to the word BEGIN and the closing brace } to the word END. Once these symbols are equated the words BEGIN and END can be used in place of the respective braces. This is illustrated in Program 3-12.

Program 3-12

```
#define BEGIN {
#define END }
#define PI 3.1416
#define CONVERT PI/180.0
#include <stdio.h>
#include <math.h>
main( )
BEGIN
  float angle;

  printf("Enter the angle (in degrees): ");
  scanf("%f", &angle);
  printf("\nThe sine of the angle is %f", sin(CONVERT*angle));
END
```

When Program 3-12 is compiled, the preprocessor faithfully replaces all occurrences of the words BEGIN and END with their equivalent symbols. Although using #define statements to create a new set of symbols equivalent to the standard C symbol set usually is not a good idea, Program 3-12 should give you an idea of the richness and diversity that C provides. Generally, the constructions that can be created in C are limited only by the imagination and good sense of the programmer.

Exercises 3.4

1. Modify Program 3-1 to use the symbolic constant GRAV in place of the value 32.2 used in the program. Compile and execute your program to verify that it produces the same result shown in the text.

2. Rewrite the following program using a `#define` statement for the constant `3.1416`:

```
#include <stdio.h>
#include <math.h>
main( )
{
  float radius, circum, area;

  printf("\nEnter a radius: ");
  scanf("%f", &radius);
  circum = 2.0 * 3.1416 * radius;
  area = 3.1416 * pow(radius,2.0);
  printf("The circumference of the circle is %f", circum);
  printf("\nThe area of the circle is %f", area);
}
```

3. Rewrite the following program so that the variable `prime` is changed to a symbolic constant:

```
#include <stdio.h>
#include <math.h>
main( )
{
  float prime, amount, interest;

  prime = 0.08;       /* prime interest rate */
  printf("\nEnter the amount: ");
  scant("%f", &amount);
  interest = prime * amount;
  printf("\nThe interest earned is %f dollars", interest);
}
```

4. Rewrite the following program to use the symbolic constant `FACTOR` in place of the expression `(5.0/9.0)` used in the program:

```
#include <stdio.h>
#include <math.h>
main( )
{
  float fahren, celsius;

  printf("\nEnter a temperature in degrees Fahrenheit: ");
  scanf("%f", &fahren);
  celsius = (5.0/9.0) * (fahren - 32.0);
  printf("\nThe equivalent Celsius temperature is %f", celsius);
}
```

3.5 Applications

In this section we present two applications to further illustrate both the use of `scanf( )` function calls to accept user input data and the use of library functions for performing calculations.

Application 1: Acid Rain

The use of coal as the major source of steam power began with the Industrial Revolution. Currently coal is one of the principal sources of generating electrical power in many industrialized countries.

Since the middle of the 19th century it has been known that the oxygen used in the burning process combines with the carbon and sulfur in the coal to produce both carbon dioxide and sulfur dioxide. When these gases are released into the atmosphere the sulfur dioxide combines with the water and oxygen in the air to form sulfuric acid, which itself is transformed into separate hydronium ions and sulfates (see Figure 3–8). The hydronium ions in the atmosphere fall to earth either as components of rain or as a dry deposition and change the acidity level of lakes and forests.

The acid level of rain and lakes is measured on a pH scale using the formula

$$pH = -\text{Log}_{10}(\text{concentration of hydronium ions})$$

where the concentration of hydronium ions is measured in units of moles/liter. A pH value of 7 indicates a neutral value (neither acid nor alkaline), while levels below 7 indicate the presence of an acid and levels above 7 indicate the presence of an alkaline substance. For example, sulfuric acid has a pH value of approximately 1, lye has a pH value of approximately 13, and water typically has a pH value of 7. Marine life usually cannot survive in water with a pH level below 4.

Using the formula for pH, we will write a C program that calculates the pH level of a substance based on a user-input value for the concentration of hydronium ions.

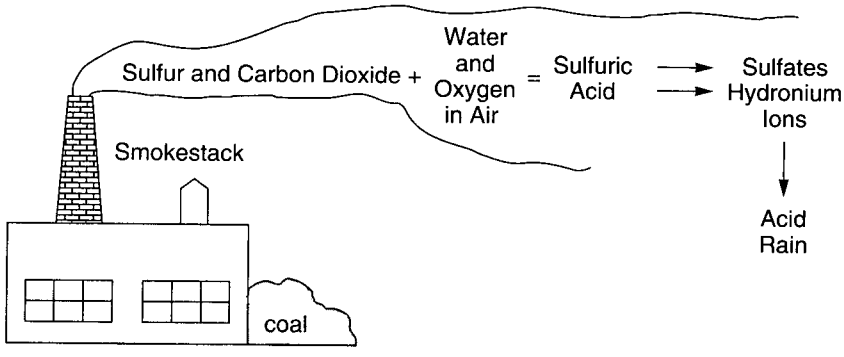
Program Development

We will use the top-down development procedure described in Chapter 2.

Step 1: Determine the Desired Outputs Although the statement of the problem provides technical information on the composition of acid rain, from a programming viewpoint this is a rather simple problem. Here there is only one required output—a pH level.

Step 2: Determine the Input Items For this problem there is only one input item, the concentration level of hydronium ions.

FIGURE 3–8 The Formation of Acid Rain



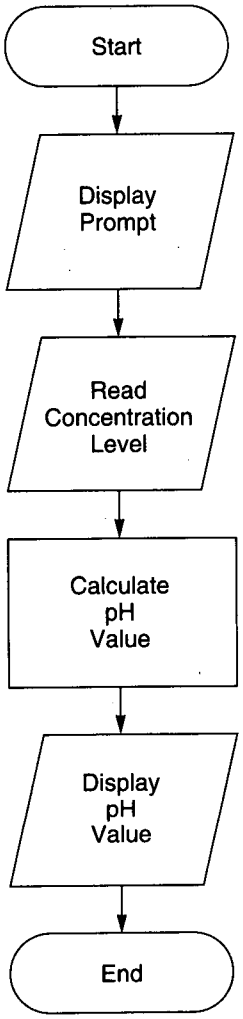
Step 3a: Determine an Algorithm The processing required to transform the input to the required output is a rather straightforward use of the pH formula that is provided. The flowchart representation of the complete algorithm for entering the input data, processing the data to produce the desired output, and displaying the output is illustrated in Figure 3–9.

The pseudocode representation of the algorithm depicted in Figure 3–9 is:

Display a prompt to enter an ion concentration level
Read a value for the concentration level
Calculate a pH level using the given formula
Display the calculated value

Step 3b: Do a Hand Calculation To ensure that we understand the formula used in the algorithm we will do a hand calculation. The result of this calculation can then be used to verify the result produced by the program. Assuming a hydroni-

FIGURE 3–9 The Selected Algorithm as a Flowchart



um concentration of .0001 (any value would do), the pH level is calculated as $-\text{Log}_{10}\ 10^{-4}$. Either by knowing that the logarithm of 10 raised to a power is the power itself, or by using a log table, the value of this expression is $-(-4) = 4$.

Step 3c: Select Variable Names The last step required before the selected algorithm is described in C is to select variable named for the input and output, and any intermediate variables required by the algorithm. For this problem we will select the variable name `hydron` for the input variable and the name `phlevel` for the calculated pH level (any valid variable name could have been selected).

Step 4: Write the Program Program 3-13 describes the selected algorithm in C using the variable names chosen in Step 3c.

Program 3-13

```
#include <stdio.h>
#include <math.h>
main( )
{
  float hydron, phlevel;

  printf("Enter the hydronium ion concentration: ");
  scanf("%f", &hydron);
  phlevel = -log10(hydron);
  printf("\nThe pH level is %f", phlevel);
}
```

Program 3-13 begins with an `#include` preprocessor statement, followed by the function `main( )`. Within `main( )`, a declaration statement declares two floating point variables, `hydron` and `phlevel`. The program then displays a prompt requesting input data from the user. After the prompt is displayed a `scanf( )` function call is used to store the entered data in the variable `hydron`. Finally, a value for `phlevel` is calculated, using the logarithmic library function, and displayed. As always, the program is terminated with a closing brace.

Step 5: Test the Program A test run using Program 3-13 produced the following:

```
Enter the hydronium ion concentration level: 0.0001
The pH level is 4.000000
```

As the program performs a single calculation, and the result of this test run agrees with our previous hand calculation, the program has been completely tested. It can now be used to calculate the pH level of other hydronium concentrations with confidence that the results being produced are accurate.

Application 2: Sine Approximation

The sine of an angle x, in radians, can be approximated using the polynomial

$$x - \frac{x^3}{6} + \frac{x^5}{120}$$

Using this polynomial as a base, write a program that approximates the sine of a user-entered angle using the first, second, and third terms, respectively, of the approximating polynomial. For each approximation display the value calculated by C's intrinsic sine function, the approximate value, and the absolute difference between the two. Make sure to verify your program using a hand calculation. Once the verification is complete, use the program to approximate the sine of 62.5 degrees.

Program Development

We will use the top-down development procedure described in Chapter 2.

Step 1: Determine the Desired Outputs This program requires a total of nine outputs, which we arrive at as follows. The statement of the problem specifies that three approximations are to be made, using one, two, and three terms of the approximating polynomial, respectively. For each approximation to the sine three output values are required: the value of the sine produced by the intrinsic `SIN( )` function, the approximated value, and the absolute difference between the two values. Figure 3–10 illustrates, in symbolic form, the structure of the required output display.

The output indicated in Figure 3–10 can be used to get a "feel" for what the program must look like. Because each line in the display can only be produced by executing a `printf( )` function call, clearly three such statements must be executed. Additionally, since each output line contains three computed values, each `printf( )` function call will have three items in its expression list.

Step 2: Determine the Input Items The only input to the program consists of the angle whose sine is to be approximated. This will, of course, require a single prompt and a `scanf( )` function call to input the necessary value.

Step 3a: Determine an Algorithm Before any output item can be calculated, it will be necessary to have the program prompt the user for an input angle and then have the program accept a user-entered value. As this angle will be in degrees and both the `SIN( )` function and the approximating polynomial require radian measure, the input angle will have to be converted into radians.

The actual output display consists of two title lines followed by three lines of calculated data. The title lines can be produced using two `printf( )` function calls. Now let's see how the actual data being displayed is produced.

The first item on the first data output line illustrated in Figure 3–10, the sine of the angle, can be obtained using the `SIN( )` function. The second item on this line, the approximation to the sine, can be obtained by using the first term in the polynomial that was given in the program specification. Finally, the third item on the line can be calculated using the `fabs( )` function on the difference between the first two items. When all of these items are calculated a single `printf( )` statement can be used to display the three results on the same line.

The second output line illustrated in Figure 3–10 displays the same type of items as the first line, except that the approximation to the sine requires using two terms of the approximating polynomial. Notice also that the first item on the second line, the sine of the angle, is the same as the first item on the first line. This means that this item does not have to be recalculated and the value calculated for the first line can simply be displayed a second time. Once the data for the second line has been calculated a single `printf( )` statement can be used to display the required values.

Finally, only the second and third items on the last output line shown in

Sine	Approximation	Difference
intrinsic function value	1st approximate value	1st difference
intrinsic function value	2nd approximate value	2nd difference
intrinsic function value	3rd approximate value	3rd difference

FIGURE 3–10 Required Output Display

Figure 3–10 need to be recalculated, since the first item on this line is the same as previously calculated for the first line. The second item on the third line is calculated using all three terms of the polynomial and the last item on this line is obtained as the absolute difference between the first two items. Once the data for this last line has been calculated a single `printf( )` statement can be used to display the complete line.

Thus, for this problem, the complete algorithm described in pseudocode is:

Display a prompt for the input angle
Read an angle in degrees
Convert the angle to radian measure
Display the heading lines
Calculate the sine of the angle
Calculate the first approximation
Calculate the first difference
Print the first output line
Calculate the second approximation
Calculate the second difference
Display the second output line
Calculate the third approximation
Calculate the third difference
Display the third output line.

Step 3b: Do a Hand Calculation To ensure that we understand the processing used in the algorithm we will do a hand calculation. The result of this calculation can then be used to verify the result produced by the program that we write. For test purposes any angle will do and we will arbitrarily select a value of 30 degrees. Converting this angle to radian measure requires multiplying it by the factor 3.1416/180.0, which corresponds to 0.523600 radians. For this test value the following approximations to the sine are obtained:

Using the first term of the polynomial the approximation is:

$$0.523600$$

Using the first two terms of the polynomial the approximation is:

$$0.523600 - (0.523600)^3 / 6 = .499675$$

Using all three terms of the polynomial the approximation is:

$$0.499675 + (0.523600)^5 / 120 = .500003$$

Notice that in using all three terms of the polynomial it was not necessary to recalculate the value of the first two terms. Instead, we used the value previously calculated in our second approximation. All of the approximations are extremely close to 0.5, which is the actual sine of 30 degrees.

Step 3c: Select Variable Names The last step required before the selected algorithm is described in C is to select variable names for the input and output, and any intermediate variables required by the algorithm. For this problem we will select the variable name `angle` for the input angle entered by the user and the name `radian` for its equivalent radian measure. The conversion factor 3.1416/180.0 to convert the input angle from degrees into radians will be given the name `CONVERT` and constructed as a named constant using the `#define` statement. Finally, the variable names `sine`, `approx`, and `difrence` will be declared for the sine of the angle, the approximation to the sine, and the difference between these two quantities, respectively. As always any valid variable names could have been selected.

Step 4: Write the Program Program 3-14 represents a description of the selected algorithm in the C language.

In reviewing Program 3-14 notice that the input angle is immediately converted to radians after it is read. The two title lines are printed prior to any calculations being made. The value of the sine is then computed using the intrinsic sine function and assigned to the variable sine. This assignment permits this value to be used in the three difference calculations and displayed three times without the need for recalculation.

Since the approximation to the sine is "built up" using more and more terms of the approximating polynomial, only the new term for each approximation is calculated and added to the previous approximation. Finally, to permit the same variables to be used again, the values in them are immediately printed before the next approximation is made. Following is a sample run produced by Program 3-14:

```
Enter an angle (in degrees): 30.0
     SINE        APPROXIMATION     DIFFERENCE
-------------    -------------    -------------
   0.500001         0.523600         0.023599
   0.500001         0.499675         0.000326
   0.500001         0.500003         0.000002
```

Step 5: Test the Program The first two columns of output data produced by the sample run agree with our hand calculation (if you are unfamiliar with exponental notation review Section 2.1). A hand check of the last column verifies that it also correctly contains the difference in values between the first two columns.

As the program only performs seven calculations and the result of the test run agrees with our hand calculations, the program has been completely tested. It can now be used with other input angles with confidence in the results produced.

Program 3-14

```
#define CONVERT  3.1416/180.0
#include <stdio.h>
#include <math.h>
/* this program approximates the sine of an angle, in degrees */
/* using one, two, and three terms of an approximating polynomial */
main( )
{
  float angle, radian, sine, approx, difrence;

  printf("Enter an angle (in degrees): ");
  scanf("%f", &angle);
  radian = CONVERT * angle;  /* convert to radian measure */

    /* print two title lines */
  printf("\n    SINE          APPROXIMATION      DIFFERENCE");
  printf("\n-------------     -------------     ------------- ");

  sine = sin(radian);      /* use the library function for the sine */
    /* calculate the first approximation */
  approx = radian;
  difrence = fabs(sine - approx);
  printf("\n%10.6f     %13.6f     %13.6f", sine, approx, difrence);
    /* calculate the second approximation */
  approx = approx - pow(radian,3.0)/6.0;
  difrence = fabs(sine - approx);
  printf("\n%10.6f     %13.6f     %13.6f", sine, approx, difrence);
    /* calculate the third approximation */
  approx = approx + pow(radian,5.0)/120.0;
  difrence = fabs(sine - approx);
  printf("\n%10.6f     %13.6f     %13.6f", sine, approx, difrence);
}
```

Additional Exercises for Chapter 3

1. Enter, compile, and run Program 3-13 on your computer system.
2. Enter, compile, and run Program 3-14 on your computer system.
3. By mistake a student wrote Program 3-14 as follows:

```
#include <math.h>
#define CONVERT  3.1416/180.0
/* this program approximates the sine of an angle, in degrees */
```

```
/* using one, two, and three terms of an approximating polynomial */
main( )
{
  float angle, radian, sine, approx, difrence;

  printf("\n   SINE          APPROXIMATION      DIFFERENCE");
  printf("\n------------     -------------     ------------ ");
  printf("Enter an angle (in degrees): ");
  scanf("%f", &angle);
  radian = CONVERT * angle;  /* convert to radian measure */
  sine = sin(radian);       /* use the library function for the sine */
  approx = radian;          /* first approximation */
  difrence = fabs(sine - approx);
  printf("\n%10.6f     %13.6f     %13.6f", sine, approx,
difrence);
  approx = approx - pow(radian,3)/6.0;    /* second approximation */
  difrence = fabs(sine - approx);
  printf("\n%10.6f     %13.6f     %13.6f", sine, approx,
difrence);
  approx = approx + pow(radian,5)/120.0;  /* third approximation */
  difrence = fabs(sine - approx);
  printf("\n%10.6f     %13.6f     %13.6f", sine, approx,
difrence);
}
```

Determine the output that will be produced by this program.

4 a. The formula for the standard normal deviate, z, used in statistical applications is: where μ refers to a mean value and σ to a standard deviation.

$$z = \frac{X - \mu}{\sigma}$$

Using this formula, write a program that calculates and displays the value of the standard normal deviate when X = 85.3, μ = 80, and σ = 4.

b. Rewrite the program written in Exercise 4a to accept the values of *x*, μ, and σ as user inputs while the program is executing.

5 a. The equation of the normal (bell-shaped) curve used in statistical applications is:

$$y = \frac{1}{\sigma\sqrt{2\pi}} e^{-\left[(1/2)(x-\mu)/\sigma\right]^2}$$

Using this equation, and assuming μ = 90 and σ = 4, write a program that determines and displays the value of y when x = 80.

b. Rewrite the program written in Exercise 5a to accept the values of *x*, μ, and σ as user inputs while the program is executing.

6 a. Write, compile, and execute a program that calculates and displays the gross pay and net pay of two individuals. The first individual works 40 hours and is paid an hourly rate of $8.43. The second individual works 35 hours and is paid an hourly rate of $5.67. Both individuals have 20 percent of their gross pay withheld for income tax purposes and both pay 2 percent of their gross pay, before taxes, for medical benefits.

b. Redo Exercise 6a assuming that the individuals' hours and rate will be entered when the program is run.

7. The volume of oil stored in a underground 200-foot-deep cylindrical tank is determined by measuring the distance from the top of the tank to the surface of the oil. Knowing this distance and the radius of the tank, the volume of oil in the tank can be determined using the formula $volume = \pi\ radius^2\ (200 - distance)$. Using this information, write, compile, and execute a C program that accepts the radius and distance measurements, calculates the volume of oil in the tank, and displays the two input values and the calculated volume. Verify the results of your program by doing a hand calculation using the following test data: radius equals 10 feet and distance equals 12 feet.

8. The perimeter, surface area, and volume of an in-ground pool are given by the following formulas:

Perimeter = 2(*length* + *width*)
Volume = *length* * *width* * *average depth*
Underground surface area = 2(*length* + *width*) *average depth* + *length* * *width*

Using these formulas as a basis, write a C program that accepts the length, width, and average depth measurements, and calculates the perimeter, volume, and underground surface area of the pool. In writing your program make the following two calculations immediately after the input data has been entered: *length* * *width* and *length* + *width*. The results of these two calculations should then be used, as appropriate, in the assignment statements for determining the perimeter, volume, and underground surface area. Verify the results of your program by doing a hand calculation using the following test data: length equals 25 feet, width equals 15 feet, and average depth equals 5.5 feet. When you have verified that your program is working, use it to complete the following table:

Length	Width	Depth	Perimeter	Volume	Underground Surface Area
25	10	5.0			
25	10	5.5			
25	10	6.0			
25	10	6.5			
30	12	5.0			
30	12	5.5			
30	12	6.0			
30	12	6.5			

3.6 Common Programming Errors

In using the material presented in this chapter, be aware of the following possible errors:

1. Forgetting to assign initial values to all variables before the variables are used in an expression. Initial values can be assigned when the variables are declared, by explicit assignment statements, or by interactively entering values using the `scanf( )` function.
2. Using a library function without including the preprocessor statement `#include <math.h>`.

3. Using a library function without providing the correct number or arguments having the proper data type.
4. Forgetting to pass addresses to `scanf( )`. Since `scanf( )` treats all arguments following the control string as addresses, it is up to the programmer to ensure that addresses are passed correctly.
5. Including a message within the control string passed to `scanf( )`. Unlike `printf( ), scanf( )`'s control string typically contain only conversion control sequences.
6. Not including the correct conversion control sequences in `scanf( )` function calls for the data values that must be entered.
7. Not closing off the control string passed to `scanf( )` with a double quote symbol followed by a comma, and forgetting to separate all arguments passed to `scanf( )` with commas.
8. Terminating `#include` and `#define` statements to the preprocessor with a semicolon. By now you probably end every line in your C programs with a semicolon, almost automatically. But there are cases, for example preprocessor commands, where a semicolon should not end a line.
9. Rushing to code and run a program before fully understanding what is required and the algorithms and procedures that will be used to produce the desired result. A symptom of this haste to get a program entered into the computer is the lack of either an outline of the proposed program or a written program itself (see the Enrichment Section at the end of this chapter). Many problems can be caught just by checking a copy of the program, either handwritten or listed from the computer, before it is compiled.
10. Being unwilling to test a program in depth. After all, since you wrote the program you assume it is correct or you would have changed it before it was compiled. It is extremely difficult to back away and honestly test your own software. As a programmer you must constantly remind yourself that just because you think your program is correct does not make it so. Finding errors in your own program is a sobering experience, but one that will help you become a master programmer.

3.7 Chapter Summary

1. C provides library functions for calculating trigonometric, logarithmic, and other mathematical computations typically required in scientific and engineering programs. Each program using one of these mathematical functions must either include the statement `#include <math.h>` or have a function declaration for the mathematical function before it is called.
2. Data passed to a function are called *arguments* of the function. Arguments are passed to a library function by including each argument, separated by commas, within the parentheses following the function's name. Each function has its own requirements for the number and data types of the arguments that must be provided.
3. Every mathematical library function operates on its arguments to calculate a single value. To use a library function effectively you must know what the

function does, the name of the function, the number and data types of the arguments expected by the function, and the data type of the returned value.

4. Functions may be included within larger expressions.
5. The `scanf( )` function is a standard library function used for data input. `scanf( )` requires a control string and a list of addresses. The general form of this function call is:

```
scanf("control string", &arg1, &arg2, . . . , &argn);
```

The control string typically contains only conversion control sequences, such as `%d`, and must contain the same number of conversion control sequences as argument addresses.

6. When a `scanf( )` function call is encountered the computer temporarily suspends further statement execution until sufficient data has been entered for the number of variables contained in the `scanf( )` function.
7. It is good programming practice to display a message, prior to a `scanf( )` function call, that alerts the user as to the type and number of data items to be entered. Such a message is called a *prompt*.
8. Each compiled C program is automatically passed through a preprocessor. Lines beginning with # in the first column are recognized as commands to this preprocessor. Preprocessor commands are not terminated with a semicolon.
9. Expressions can be made equivalent to a single identifier using the preprocessor `define` command. This command has the form

```
#define  identifier  expression-or-text
```

it allows the identifier to be used instead of the expression or text anywhere in the program after the command. Generally, a `define` command is placed at the top of a C program.

3.8 Enrichment Study: Program Life Cycle

Just as people and products have a life cycle, so do programs. A program's life cycle is divided into three main stages as illustrated in Figure 3–11. These stages consist of program development, program documentation, and program maintenance.

The development stage is where a program is initially developed. It is at this stage that requirements must be understood and the structure of the program is planned using the top-down development procedure presented in Section 2.5. The documentation stage, as its name implies, consists of creating, both within the program and in separate documents, sufficient user and programmer support references and explanations. At the maintenance stage the program is modified or enhanced as new demands and requirements are obtained or program errors are detected. Complete courses and textbooks are devoted to each of these three program stages. Our purpose in listing them is to put the actual writing of a program in perspective with the total effort needed to produce professional engineering and scientific software.

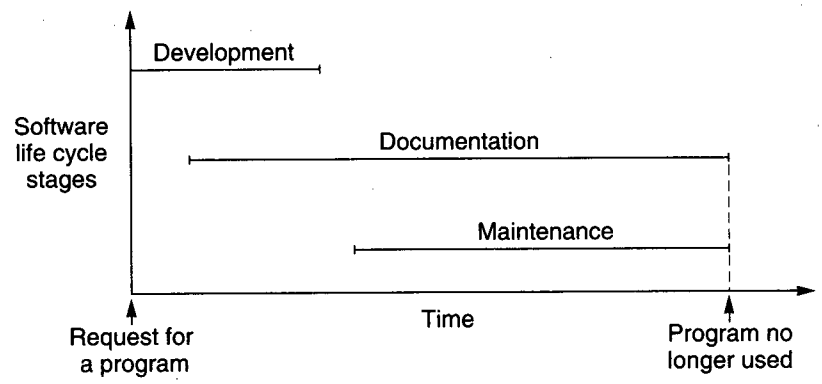


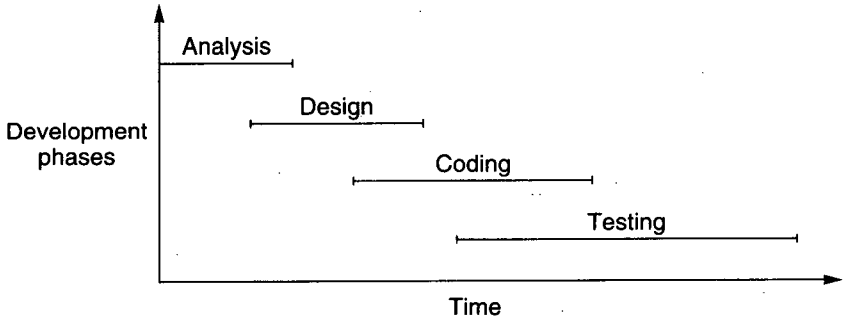
FIGURE 3–11 A Program's Life Cycle

The writing of a program in a computer language is formally called coding (informally, of course, it is called programming). And that, after all, is what we have been doing—writing programs in a language, or code, that can be decoded and used by the computer. As we saw in Section 2.5, the coding of a program is but one component in the program's development stage. The total development effort is composed of four distinct phases, as illustrated in Figure 3–12.

Listed below are both the steps in the top-down development procedure corresponding to each development phase and the relative amount of effort that is typically expended on each phase in large engineering and scientific programming projects. As can be seen from this listing, the coding phase is not the major effort in overall program development.

Phase	Top-Down Development Step	Effort
Analysis	Steps 1 and 2	10%
Design	Step 3	20%
Coding	Step 4	20%
Testing	Step 5	50%

FIGURE 3–12 The Phases of Program Development



Many new programmers have trouble because they spend the majority of their time coding the program, without spending sufficient time understanding and designing the program. In this regard, it is worthwhile to remember the programming proverb, "It is impossible to write a successful program for a problem or application that is not fully understood."

For this reason the analysis phase is one of the most important, because if the requirements are not fully and completely understood before programming begins, the results are almost always disastrous. Once a program structure is created and the program is written, new or reinterpreted requirements often cause havoc. An analogy with house construction is useful to illustrate this point.

Imagine designing and building a house without fully understanding the architect's specifications. After the house is completed, the architect tells you that a bathroom is required on the first floor, where you have built a wall between the kitchen and the dining room. In addition, that particular wall is one of the main support walls for the house and contains numerous pipes and electrical cables. In this case, adding one bathroom requires a major modification to the basic structure of the house.

Experienced programmers understand the importance of analyzing and understanding a program's requirements before coding, if for no other reason than that they too have constructed programs that later had to be entirely dismantled and redone. The following exercise should give you a sense of this experience.

Figure 3–13 illustrates the outlines of six individual shapes from a classic children's puzzle. Assume that as one or more shapes are given, starting with shapes A and B, an easy-to-describe figure must be constructed.

Typically, shapes A and B are initially arranged to obtain a square, as illustrated in Figure 3–14. Next, when shape C is considered, it is usually combined with the existing square to form a rectangle, as illustrated in Figure 3–15. Then when pieces D and E are added, they are usually arranged to form another rectangle, which is placed alongside the existing rectangle to form a square, as shown in Figure 3–16.

The process of adding new pieces onto the existing structure is identical to constructing a program and then adding to it as each subsequent requirement is understood. The problem arises when the program is almost finished and a requirement is added that does not fit easily into the established pattern. For example, assume that the last shape (shape F) is now to be added (see Figure 3–17). This last piece does not fit into the existing pattern that has been constructed. In order to include this piece with the others, the pattern must be completely dismantled and restructured.

Unfortunately, many programmers structure their programs in the same manner used to construct Figure 3–16. Rather than taking the time to understand the complete set of requirements, new programmers frequently start coding based on their understanding of only a small subset of the total requirements. Then, when a subsequent requirement does not fit the existing program structure, the programmer is forced to dismantle and restructure either parts or all of the program.

Now, let's approach the problem of creating a figure from another view. If we started by arranging the first set of pieces as a parallelogram, all the pieces could be included in the final figure, as illustrated in Figure 3–18.

It is worthwhile observing that the piece that caused us to dismantle the first

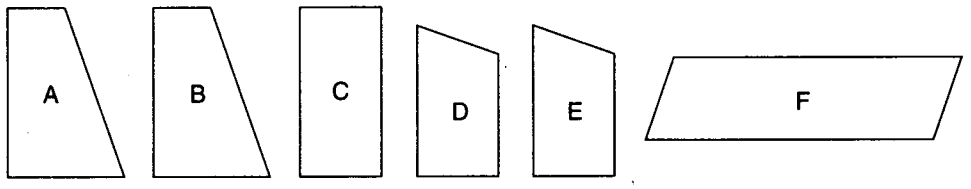


FIGURE 3–13 Six Individual Shapes

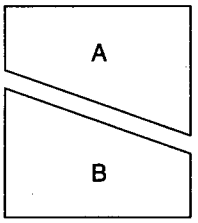


FIGURE 3–14 Typical First Figure

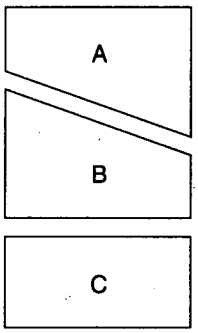


FIGURE 3–15 Typical Second Figure

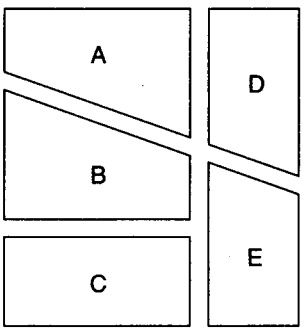


FIGURE 3–16 Typical Third Figure

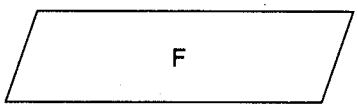


FIGURE 3–17 The Last Piece

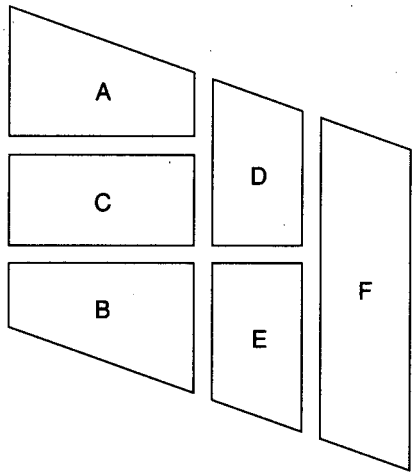


FIGURE 3–18 Including All the Pieces

figure (Figure 3–16) actually sets the pattern for the final figure illustrated in Figure 3–18. This is often the case with programming requirements. The requirement that seems to be the least clear is frequently the one that determines the main interrelationships of the program. Thus, it is essential to include and understand all the known requirements before coding is begun. In practical terms this means doing the analysis and design before any coding is attempted.

Selection

Chapter Four

4.1 Relational Expressions
4.2 The `if-else` Statement
4.3 Nested `if` Statements
4.4 The `switch` Statement
4.5 Applications
4.6 Common Programming Errors
4.7 Chapter Summary
4.8 Enrichment Study: A Closer Look at Errors, Testing, and Debugging

The field of programming, as a distinct discipline, is still relatively new. It should not be surprising, then, that many advances have occurred in the theoretical foundations of this field. One of the most important of these advances was the recognition in the late 1960s that any algorithm, no matter how complex, could be constructed using combinations of three standardized *flow of control* structures: sequential, selection, and repetition.

The term *flow of control* refers to the order in which a program's statements are executed. Unless directed otherwise, the normal flow of control for all programs is *sequential.* This means that statements are executed in sequence, one after another, in the order in which they are placed in the program.

Selection and repetition structures permit the sequential flow of control to be altered in precisely defined ways. As you might have guessed, the *selection* structure is used to select which statements are to be performed next and the *repetition* structure is used to repeat a set of statements. In this chapter we present C's selection statements. As selection requires choosing between alternatives, we begin this chapter with a description of C's selection criteria.

4.1 Relational Expressions

Besides providing arithmetic capabilities (addition, subtraction, multiplication, division, etc.) all computers have logical capabilities that permit comparisons of various quantities. Because many seemingly "intelligent" decision-making situations can be reduced to the level of choosing between two values, a computer's comparison capability can be used to create a remarkable intelligence like facility.

The expressions used to compare operands are called *relational expressions.* A simple relational expression consists of a relational operator connecting two variable and/or constant operands, as shown in Figure 4–1. The relational operators available in C are given in Table 4–1. These relational operators may be used with integer, float, double, or character operands, but must be typed exactly as given in Table 4–1. Thus, while the following examples are all valid:

```
age > 40          length <= 50          temp > 98.6
  3 < 4           flag == done          id_num == 682
day != 5          2.0 > 3.3             hours > 40
```

the following are invalid:

```
length =< 50      /* operator out of order  */
2.0 >> 3.3        /* invalid operator       */
flag = = done     /* spaces are not allowed */
```

Relational expressions are sometimes called *conditions,* and we will use both terms to refer to these expressions. Like all C expressions, relational expressions are evaluated to yield a numerical result.[1] In the case of relational expressions,

[1] In this regard C differs from other high-level programming languages that yeild a Boolean (true, false) result.

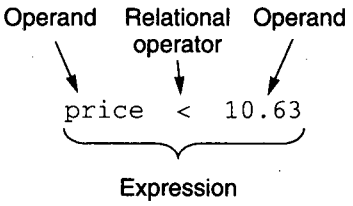


FIGURE 4–1 Anatomy of a Simple Relational Expression

the value of the expression can only be an integer value of 1 or 0, which is interpreted as true and false, respectively. *A condition that we would interpret as true evaluates to an integer value of 1, and a false condition results in an integer value of 0.* For example, because the relationship 3 < 4 is always true, this expression has a value of 1, and because the relationship 2.0 > 3.3 is always false, the value of the expression itself is 0. This can be verified using the statements

```
printf("The value of 3 > 4 is %d", 3 > 4);
printf("\nThe value of 2.0 > 3.0 is %d, 2.0 > 3.3);
```

which results in the display

```
The value of 3 < 4 is 1
The value of 2.0 > 3.0 is 0
```

The value of a relational expression such as `hours > 40` depends on the value stored in the variable `hours`.

In a C program, a relational expression's value is not as important as the interpretation C places on the value when the expression is used as part of a selection statement. In these statements, which are presented in the next section, we will see that *a zero value is used by C to represent a false condition and any nonzero*

TABLE 4–1 Relational Operators in C

Relational operator	Meaning	Example
<	less than	`age < 30`
>	greater than	`height < 6.2`
<=	less than or equal to	`taxable <= 20000`
>=	greater than or equal to	`temp >= 98.6`
==	equal to	`grade == 100`
!=	not equal to	`number != 250`

value is used to represent a true condition. The selection of which statement to execute next is then based on the value obtained.

In addition to numerical operands, character data can also be compared using relational operators. For example, in the ASCII code the letter `'A'` is stored using a code having a lower numerical value than the letter `'B'`, the code for a `'B'` is lower in value than the code for a `'C'`, and so on. For character sets coded in this manner, the following conditions are evaluated as listed below.

Expression	Value	Interpretation
`'A' > 'C'`	0	False
`'D' <= 'Z'`	1	True
`'E' == 'F'`	0	False
`'G' >= 'M'`	0	False
`'B' != 'C'`	1	True

Comparing letters is essential in alphabetizing names or using characters to select a particular choice in decision-making situations.

Logical Operators

In addition to using simple relational expressions as conditions, we can create compound conditions using the logical operators AND, OR, and NOT. These operators are represented by the respective symbols:

```
&&
||
!
```

When the AND operator, `&&`, is used with two expressions, the resulting condition is called a *logical expression* and is true only if both individual expressions are true by themselves. Thus, the logical expression

```
(voltage > 40) && (milliamp < 10)
```

is true (has a value of 1) only if `voltage` is greater than `40` and `milliamp` is less than `10`. The parentheses surrounding the individual relational expressions are used for clarity only, because the logical `&&` operator has a lower precedence than the relational operators (==, <, >, etc.).

The logical OR operator, `||`, must also be applied between two expressions, and the resulting condition is also referred to as a logical expression. When using the OR operator, the condition is true if either one or both of the two individual expressions is true. Thus, the compound condition

```
(voltage > 40) || (milliamp < 10)
```

is true if either `voltage` is greater than `40`, or `milliamp` is less than `10`, or both conditions are true. Again, the parentheses around the relational expressions are used only for clarity, since the `||` operator has a lower precedence than all relational operators.

For the declarations

```
int i,j;
float a,b,complete;
```

the following represent valid conditions:

```
a > b
i == j || a < b || complete
a/b > 5 && i <= 20
```

Before these conditions can be evaluated, the values of `a`, `b`, `i`, `j`, and `complete` must be known. Assuming

`a` = 12.0, `b` = 2.0, `i` = 15, `j` = 30, and `complete` = 0.0

the previous expressions yield the following results:

Expression	Value	Interpretation
`a > b`	1	True
`i == j || a < b || complete`	0	False
`a/b > 5 && i <= 20`	1	True

The NOT operator, `!`, is used to change any expression to its opposite state. That is, if the expression has any nonzero value (true), `!expression` produces a zero value (false). If an expression is false to begin with (has a zero value), `!expression` is true and evaluates to 1. For example, assuming the number `26` is stored in the variable `voltage`, the relational expression `voltage > 40` has a value of zero (it is false), while the expression `!(voltage > 40)` has a value of 1 (it is true). Since the NOT operator is a unary operator, it is used with only one expression.

The relational and logical operators have a hierarchy of execution similar to the arithmetic operators. Table 4–2 lists the precedence of these operators in relation to the other operators we have used.

The following example illustrates the use of an operator's precedence and associativity to evaluate relational expressions, assuming the following declarations:

```
char key = 'm';
int i = 5, j = 7, k = 12;
double x = 22.5;
```

Expression	Equivalent Expression	Value	Interpretation
`i + 2 == k -1`	`(i + 2) == (k - 1)`	0	False
`3 * i - j < 22`	`((3 * i) - j) < 22`	1	True
`i + 2 * j > k`	`(i + (2 * j)) > k`	1	True
`k + 3 <= -j + 3 * i`	`(k + 3) <= ((-j) + (3*i))`	0	False
`'a' + 1 == 'b'`	`('a' + 1) == 'b'`	1	True
`key - 1 > 'p'`	`(key - 1) > 'p'`	0	False
`key + 1 == 'n'`	`(key + 1) == 'n'`	1	True
`25 <= x + 1.0`	`25 <= (x + 1.0)`	0	False

As with all expressions, parentheses can be used to alter the assigned operator priority and improve the readability of relational expressions. By evaluating the expressions within parentheses first, the following compound condition is evaluated as:

```
(6 * 3 == 36 / 2) || (13 < 3 * 3 + 4) && !(6 - 2 < 5)
       (18 == 18) ||    (13 < 9 + 4)  && !(4 < 5)
                1 ||      (13 < 13)   && !1
                1 ||             0    && 0
                1 ||             0
                1
```

TABLE 4–2 Hierarchy of Relational and Logical

Operator	Associativity	Precedence
`! unary - ++ --`	right to left	highest
`* / %`	left to right	
`+ -`	left to right	
`< <= > >=`	left to right	
`== !=`	left to right	
`&&`	left to right	
`||`	left to right	
`= += -= *= /=`	right to left	lowest

A Numerical Accuracy Problem

A subtle numerical accuracy problem relating to floating point and double precision numbers can occur with C's relational expressions. Due to the way computers store these numbers, tests for equality of floating point and double precision values and variables using the relational operator == should be avoided.

The reason for this is that many decimal numbers, such as .1, for example, cannot be represented exactly in binary using a finite number of bits. Thus, testing such numbers for exact equality can fail. When equality of non-integer values is desired it is better to require that the absolute value of the difference between operands be less than some extremely small value. Thus, for floating point and double precision operands the general expression:

```
operand_1 == operand_2
```

should be replaced by the condition

```
fabs(operand_1 - operand_2) < 0.000001
```

where the value 0.000001 can be altered to any other acceptably small value. Thus, if the difference between the two operands is less than 0.000001 (or any other user selected amount), the two operands are considered essentially equal. For example if x and y are floating point variables, a condition such as

```
x/y == 0.35
```

should be programmed as

```
fabs(x/y - 0.35) < 0.000001
```

This latter condition ensures that slight inaccuracies in representing non-integer numbers in binary do not affect evaluation of the tested condition. Since all computers have an exact binary representation of zero, comparisons for exact equality to zero don't encounter this numerical accuracy problem.

Exercises 4.1

1. Determine the value of the following expressions. Assume a = 5, b = 2, c = 4, d = 6, and e = 3.

   a. `a > b`
   b. `a != b`
   c. `d % b == c % b`
   d. `a * c != d * b`
   e. `d * b == c * e`
   f. `a * b`
   g. `a % b * c`
   h. `c % b * a`
   i. `b % c * a`

2. Using parentheses, rewrite the following expressions to correctly indicate their order of evaluation. Then evaluate each expression assuming a = 5, b = 2, and c = 4.

   a. `a % b * c && c % b * a`
   b. `a % b * c || c % b * a`
   c. `b % c * a && a % c * b`
   d. `b % c * a || a % c * b`

3. Write relational expressions to express the following conditions (use variable names of your own choosing):
   a. a person's age is equal to 30
   b. a person's temperature is greater than 98.6
   c. a person's height is less than 6 feet
   d. the current month is 12 (December)
   e. the letter input is m
   f. a person's age is equal to 30 and the person is taller than 6 feet
   g. the current day is the 15th day of the 1st month
   h. a person is older than 50 or has been employed at the company for at least 5 years
   i. a person's identification number is less than 500 and the person is older than 55
   j. a length is greater than 2 feet and less than 3 feet

4. Determine the value of the following expressions, assuming `a = 5`, `b = 2`, `c = 4`, and `d = 5`.
   a. `a == 5`
   b. `b * d == c * c`
   c. `d % b * c > 5 || c % b * d < 7`

4.2 The `if-else` Statement

The `if-else` statement directs the computer to select a sequence of one or more instructions based on the result of a comparison. For example, the state of New Jersey has a two-level state income tax structure. If a person's taxable income is less than $20,000, the applicable state tax rate is 2 percent. For incomes exceeding $20,000, a different rate is applied. The `if-else` statement can be used in this situation to determine the actual tax based on whether the gross income is less than or equal to $20,000. The general form of the `if-else` statement is:

```
if (expression)
  statement1;
else
  statement2;
```

The expression is evaluated first. If the value of the expression is nonzero, *statement1* is executed. If the value is zero the statement after the reserved word `else` is executed. Thus, one of the two statements (either *statement1* or *statement2*, but not both) is always executed depending on the value of the expression. Notice that the tested expression must be put in parentheses and a semicolon is placed after each statement. (Do not put a semicolon after the parentheses or the reserved word `else`—the semicolons go after the ends of the statement.)

The flowchart for the `if-else` statement is shown in Figure 4–2.

As a specific example of an `if-else` statement, we construct a C program for determining New Jersey income taxes. As previously described, these taxes are assessed at 2 percent of taxable income for incomes less than or equal to $20,000. For taxable income greater than $20,000, state taxes are 2.5 percent of the income that exceeds $20,000 plus a fixed amount of $400. The expression to be tested is

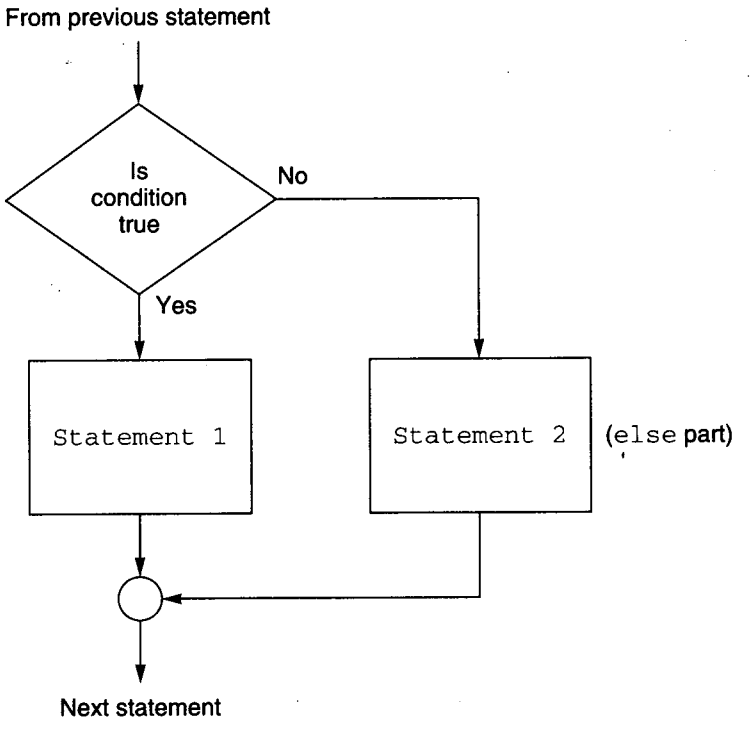


FIGURE 4–2 The **if-else** Flowchart

whether taxable income is less than or equal to $20,000. An appropriate `if-else` statement for this situation is:

```
if (taxable <= 20000.0)
  taxes = .02 * taxable;
else
  taxes = .025 * (taxable - 20000.0) + 400.0;
```

Recall that the relational operator `<=` represents the relation "less than or equal to." If the value of `taxable` is less than or equal to `20000`, the condition is true (has a value of 1) and the statement `taxes = .02 * taxable;` is executed. If the condition is not true, the value of the expression is zero, and the statement after the reserved word `else` is executed. Program 4-1 illustrates the use of this statement in a complete program.

Program 4-1

```
#include <stdio.h>
main( )
{
  float taxable, taxes;

  printf("Please type in the taxable income:");
```

```
  scanf("%f", &taxable);

  if (taxable <= 20000.0)
    taxes = .02 * taxable;
  else
    taxes = .025 * (taxable - 20000.0) + 400.0;

  printf("Taxes are $%7.2f",taxes);
}
```

A blank line was inserted before and after the `if-else` statement to highlight it in the complete program. We will continue to do this throughout the text to emphasize the statement being presented.

To illustrate selection in action, Program 4-1 was run twice with different input data. The results are:

```
Please type in the taxable income: 10000.
Taxes are $ 200.00
```

and

```
Please type in the taxable income: 30000.
Taxes are $ 650.00
```

Observe that the taxable income input in the first run of the program was less than $20,000, and the tax was correctly calculated as 2 percent of the number entered. In the second run, the taxable income was more than $20,000, and the `else` part of the `if-else` statement was used to yield a correct tax computation of

```
.025 * ($30,000. - $20,000.) + $400. = $650.
```

Although any expression can be tested by an `if-else` statement, relational expressions are predominately used. However, statements such as:

```
if (num)
  printf("Bingo!");
else
  printf("You lose!");
```

are valid. Since `num`, by itself, is a valid expression, the message `Bingo!` is displayed if `num` has any nonzero value and the message `You lose!` is displayed if `num` has a value of zero.

Compound Statements

Although only a single statement is permitted in both the `if` and `else` parts of the `if-else` statement, this statement can be a single compound statement. A *compound statement* is a sequence of single statements contained between braces, as shown in Figure 4–3.

```
{
    statement1;
    statement2;
    statement3;
        .
        .
        .
    last statement;
}
```

FIGURE 4–3 A Compound Statement Consists of Individual Statements Enclosed Within Braces

The use of braces to enclose a set of individual statements creates a single block of statements, which may be used anywhere in a C program in place of a single statement. The next example illustrates the use of a compound statement within the general form of an `if-else` statement.

```
if (expression)
{
  statement1;  /* as many statements as necessary */
  statement2;  /* can be put within the braces   */
  statement3;  /* each statement must end with a ; */
}
else
{
  statement4;
  statement5;
      .
      .
  statementn;
}
```

Program 4-2 illustrates the use of a compound statement in an actual program.

Program 4-2

```
#include <stdio.h>
main( )
{
  char temp_type;
  float temp, fahren, celsius;
  printf("Enter the temperature to be converted: ");
  scanf("%f",&temp);
  printf("Enter an f if the temperature is in Fahrenheit");
  printf("\n or a c if the temperature is in Celsius: ");
  scanf("\n%c",&temp_type); /* see footnote on page 124 */
```

```
  if (temp_type == 'f')
  {
    celsius = (5.0 / 9.0) * (temp - 32.0);
    printf("\nThe equivalent Celsius temperature is %6.2f", celsius);
  }
  else
  {
    fahren = (9.0 / 5.0) * temp + 32.0;
    printf("\nThe equivalent Fahrenheit temperature is %6.2f", fahren);
}

}
```

Program 4-2 checks whether the value in `temp_type` is `f`. If the value is `f`, the compound statement corresponding to the `if` part of the `if-else` statement is executed. Any other letter results in execution of the compound statement corresponding to the `else` part. Following is a sample run of Program 4-2.

```
Enter the temperature to be converted: 212
Enter an f if the temperature is in Fahrenheit
 or a c if the temperature is in Celsius: f

The equivalent Celsius temperature is 100.00
```

One-Way Selection

A useful modification of the `if-else` statement involves omitting the `else` part of the statement altogether. In this case, the `if` statement takes the shortened and frequently useful form:

```
if (expression)
  statement;
```

The statement following the *if (expression)* is only executed if the expression has a nonzero value (a true condition). As before, the statement may be a compound statement. The flowchart for this statement is illustrated in Figure 4–4.

This modified form of the `if` statement is called a one-way `if` statement. Program 4-3 uses this statement to selectively display a message for cars that have been driven more than 3000.0 miles.

As an illustration of its one-way selection criteria in action, Program 4-3 was run twice, each time with different input data. Only the input data for the first run causes the message `Car 256 is over the limit` to be displayed.

```
Please type in car number and mileage: 256 3562.8
  Car 256 is over the limit.
End of program output.
```

and

```
Please type in car number and mileage: 23 2562.3
End of program output.
```

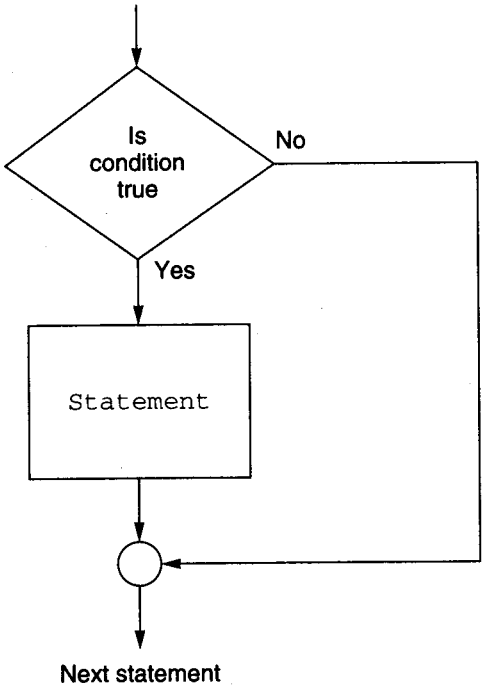


FIGURE 4–4 Flowchart for the One-Way if Statement

Program 4-3

```
#define LIMIT 3000.0
#include <stdio.h>
main()
{
  int id_num;
  float miles;

  printf("Please type in car number and mileage: ");
  scanf("%d %f", &id_num, &miles);

  if(miles > LIMIT)
    printf("  Car %d is over the limit.\n",id_num);

  printf("End of program output.\n");
}
```

Caution

Two of the most common problems encountered in initially using C's if-else statement are:

1. Misunderstanding the full implications of what an expression is.
2. Using the assignment operator, =, in place of the relational operator, ==.

Recall that an expression is any combination of operands and operators that yields a result. This definition is extremely broad and more encompassing than may be initially apparent. For example, all of the following are valid C expressions:

```
age + 5
age = 30
age == 40
```

Assuming that the variables are suitably declared, each of the above expressions yields a result. Program 4-4 uses the `printf( )` function to display the value of these expressions when age = 18.

Program 4-4

```
#include <stdio.h>
main( )
{
  int age = 18;

  printf("\nThe value of the first expression is %d", age + 5);
  printf("\nThe value of the second expression is %d", age = 30);
  printf("\nThe value of the third expression is %d", age == 40 );
}
```

The display produced by Program 4-4 is:

```
The value of the first expression is 23
The value of the second expression is 30
The value of the third expression is 0
```

As the output of Program 4-4 illustrates, each expression, by itself, has a value associated with it. The value of the first expression is the sum of the variable age plus 5, which is 23. The value of the second expression is 30, which is also assigned to the variable `age`. The value of the third expression is zero, since `age` is not equal to 40, and a false condition is represented in C with a value of zero. If the values in `age` had been 40, the relational expression `a == 40` would be true and would have a value of 1.

Now assume that the relational expression age == 40 was intended to be used in the `if` statement

```
if (age == 40)
  printf("Happy Birthday!");
```

but was mistyped as `age = 40`, resulting in

```
if (age = 40)
  printf("Happy Birthday!");
```

Since the mistake results in a valid C expression, and any C expression can be tested by an `if` statement, the resulting `if` statement is valid and will cause the message `Happy Birthday!` to be printed regardless of what value was previously assigned to age. Can you see why?

The condition tested by the if statement does not compare the value in age to the number 40, but assigns the number 40 to age. That is, the expression age = 40 is not a relational expression at all, but an assignment expression. At the completion of the assignment the expression itself has a value of 40. Since C treats any nonzero value as true, the call to printf( ) is made. Another way of looking at this is to realize that the if statement is equivalent to the following two statements:

```
age = 40;    /* assign 40 to age */
if (age)     /* test the value of age */
  printf("Happy Birthday!");
```

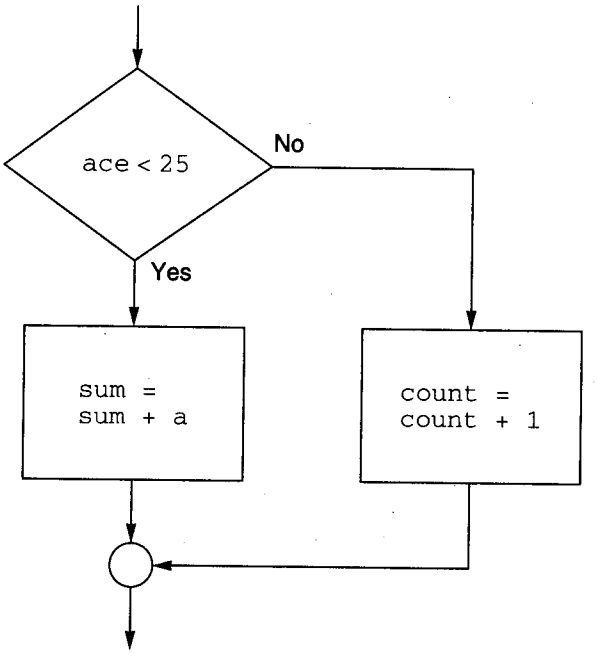
Since a C compiler has no means of knowing that the expression being tested is not the desired one, you must be especially careful when writing conditions.

Exercises 4.2

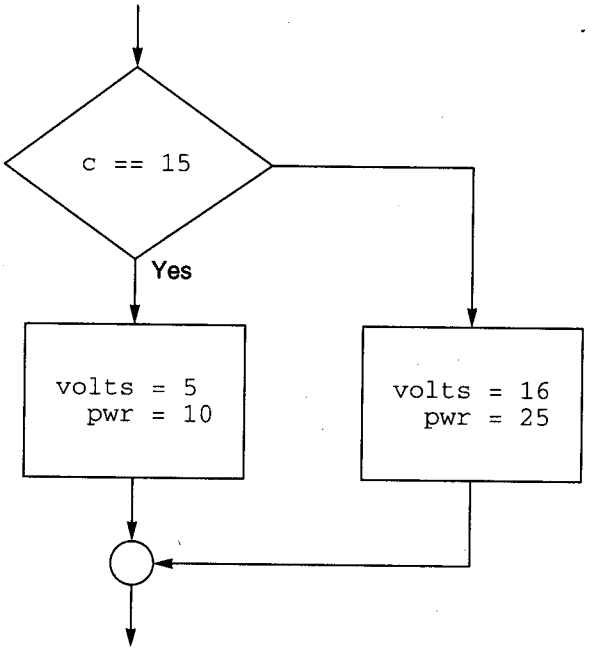
1. Write appropriate `if` statements for each of the following conditions:
   a. If `angle` is equal to 90 degrees print the message `The angle is a right angle`, else print the message `The angle is not a right angle`.
   b. If the temperature is above 100 degrees display the message `above the boiling point of water`, else display the message `below the boiling point of water`.
   c. If the number is positive add the number to `possum`, else add the number to `negsum`.
   d. If the voltage is less than .5 volts set the variable `flag` to zero, else set `flag` to one.
   e. If the difference between `volts1` and `volts2` is less than .001, set the variable `approx` to zero, else calculate `approx` as the quantity `(volts1 - volts2) / 2.0`.
   f. If the frequency is above 60 hertz, display the message `Frequency is too high`.
   g. If the difference between `temp1` and `temp2` exceeds 2.3 degrees, calculate `error` as `(temp1 - temp2) * factor`.
   h. If `x` is greater than `y` and `z` is less than 20, read in a value for `p`.
   i. If `distance` is greater than 20 and less than 35, read in a value for `time`.

2. Write `if` statements corresponding to the conditions illustrated by each of the following flowcharts.

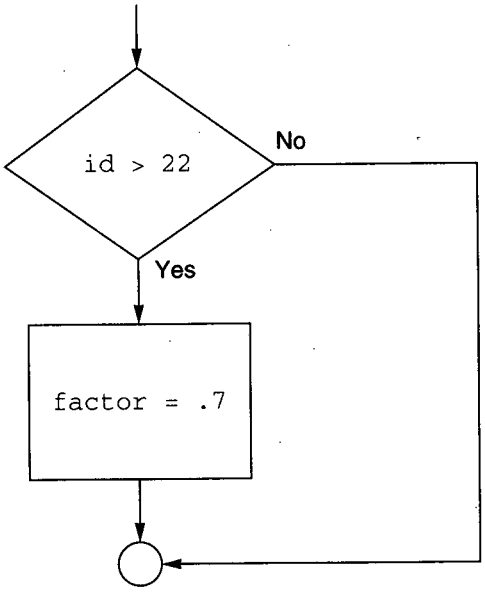
a.


ace < 25
No
Yes
sum =
sum + a
count =
count + 1

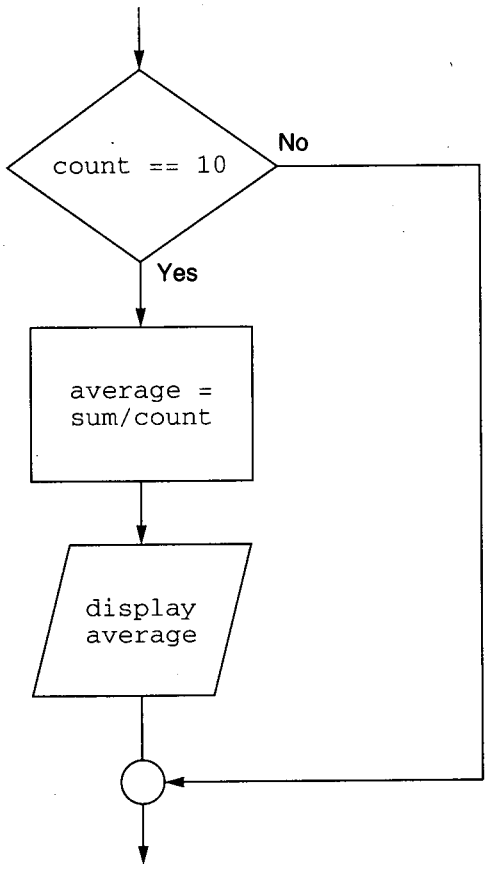
b.


c == 15
Yes
volts = 5
pwr = 10
volts = 16
pwr = 25

c.



d.



3. Write a C program that prompts the user to type in a voltage. If the entered voltage is greater than 12, the program should print the message `Battery is charging`, otherwise the program should print the message `Battery is discharging`.

4. Write a C program that asks the user to input two numbers. If the first number

entered is greater than the second number the program should print the message `The first number is greater`, else it should print the message `The first number is smaller`. Test your program by entering the numbers 5 and 8 and then using the numbers 11 and 2. What do you think your program will display if the two numbers entered are equal? Test this case.

5a. If money is left in a particular bank for more than two years, the interest rate given by the bank is 8.5 percent, else the interest rate is 7 percent. Write a C program that uses the `scanf( )` function to accept the number of years into the variable `nyrs` and display the appropriate interest rate depending on the input value.
b. How many runs should you make for the program written in Exercise 5a to verify that it is operating correctly? What data should you input in each of the program runs?

6a. In a pass/fail course, a student passes if the grade is greater than or equal to 70 and fails if the grade is lower. Write a C program that accepts a grade and prints the message `A passing grade` or `A failing grade`, as appropriate.
b. How many runs should you make for the program written in Exercise 6a to verify that it is operating correctly? What data should you input in each of the program runs?

7a. Write a C program to compute and display a person's weekly salary as determined by the following conditions: If the hours worked are less than or equal to 40, the person receives $8.00 per hour; else the person receives $320.00 plus $12.00 for each hour worked over 40 hours. The program should request the hours worked as input and should display the salary as output.
b. How many runs should you make for the program written in Exercise 7a to verify that it is operating correctly? What data should you input in each of the program runs?

8a. Write a program that displays either the message `I FEEL GREAT TODAY!` or `I FEEL DOWN TODAY #$*!` depending on the input. If the character `u` is entered in the variable code, the first message should be displayed; else the second message should be displayed.
b. How many runs should you make for the program written in Exercise 8a to verify that it is operating correctly? What data should you input in each of the program runs?

9a. A senior engineer is paid $1000 a week and a junior engineer $600 a week. Write a C program that accepts as input an engineer's status in the character variable `status`. If `status` equals `'s'`, the senior person's salary should be displayed, else the junior person's salary should be output.
b.How many runs should you make for the program written in Exercise 9a to verify that it is operating correctly? What data should you input in each of the program runs?

10. Write a C program that accepts a character as input data and determines if the character is an uppercase letter. An uppercase letter is any character that is greater than or equal to `'A'` and less than or equal to `'Z'`. If the entered character is an uppercase letter, display the message `The character just entered is an uppercase letter`. If the entered letter is not uppercase, display the message `The character just entered is not an uppercase letter`.

11. Repeat Exercise 10 to determine if the character entered is a lowercase letter. A lowercase letter is any character greater than or equal to `'a'` and less than or equal to `'z'`.

12. The following program displays the message `Hello there!` regardless of the letter input. Determine where the error is.

```
#include <stdio.h>
main( )
{
  char letter;

  printf("Enter a letter: ");
  scanf("%c",&letter);
  if (letter = 'm') printf("Hello there!");
}
```

4.3 Nested if Statements

As we have seen, an if-else statement can contain simple or compound statements. Any valid C statement can be used, including another if-else statement. Thus, one or more if-else statements can be included within either part of an if-else statement. The inclusion of one or more if statements within an existing if statement is called a *nested* if statement. For example, substituting the one-way if statement

```
if (distance < 500)
  printf("snap");
```

for statement1 in the following if statement

```
if (hours < 9)
  statement1;
else
  printf("pop");
```

results in the nested if statement

```
if (hours < 9)
{
  if (distance > 500)
    printf("snap");
}
else
  printf("pop");
```

The braces around the inner one-way if are essential, because in their absence C associates an else with the closest unpaired if. Thus, without the braces, the above statement is equivalent to

```
if (hours < 9)
  if (distance > 500)
    printf("snap");
  else
    printf("pop");
```

Here the else is paired with the inner if, which destroys the meaning of the original if-else statement. Notice also that the indentation is irrelevant as far as the compiler is concerned. Whether the indentation exists or not, *the statement is compiled by associating the last* else *with the closest unpaired if, unless braces are used to alter the default pairing.*

The process of nesting if statements can be extended indefinitely, so that the printf("snap"); statement could itself be replaced by either a complete if-else statement or another one-way if statement.

Figure 4–5 illustrates the general form of a nested if-else statement when

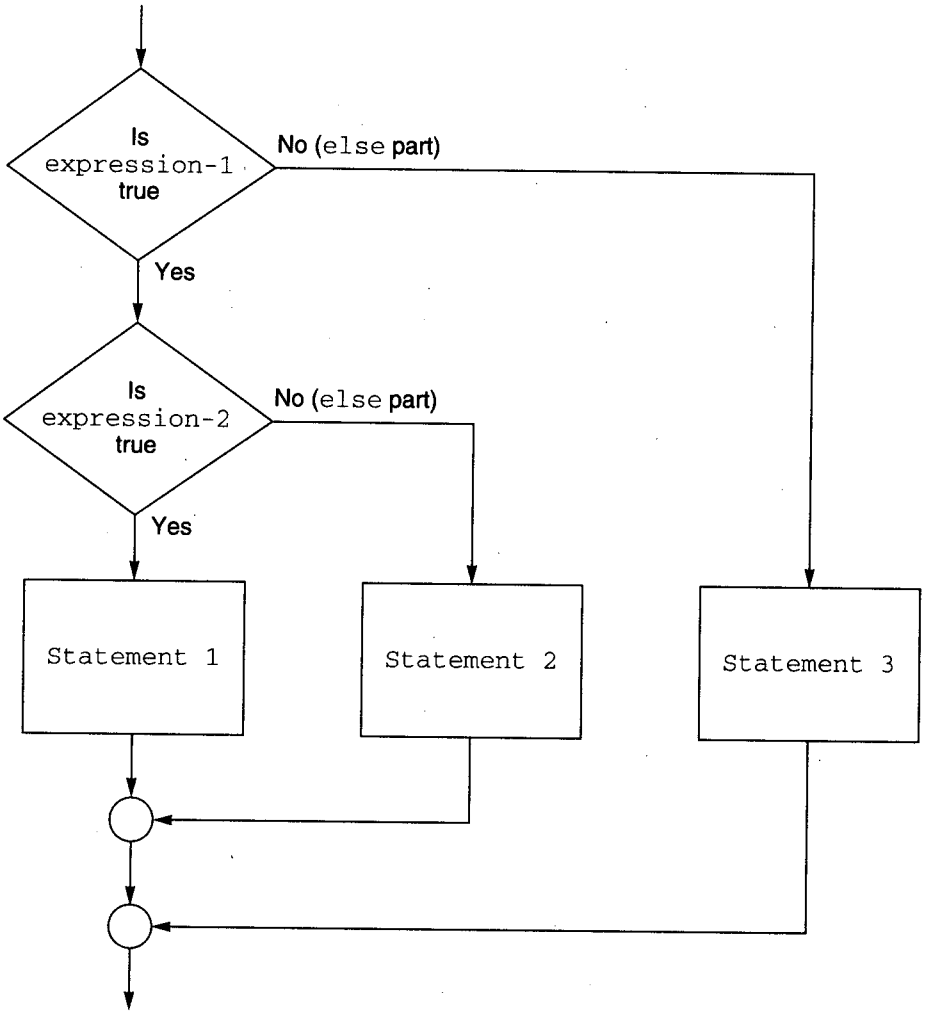


FIGURE 4–5a `if-else` Nested within `if`

an `if-else` statement is nested (a) within the `if` part of an `if-else` statement and (b) within the `else` part of an `if-else` statement.

The `if-else` Chain

In general, the nesting illustrated in Figure 4–5a tends to be confusing and is best avoided in practice. However, an extremely useful construction occurs for the nesting illustrated in Figure 4–5b, which has the form:

```
if (expression_1)
  statement1;
else
  if (expression_2)
    statement2;
  else
    statement3;
```

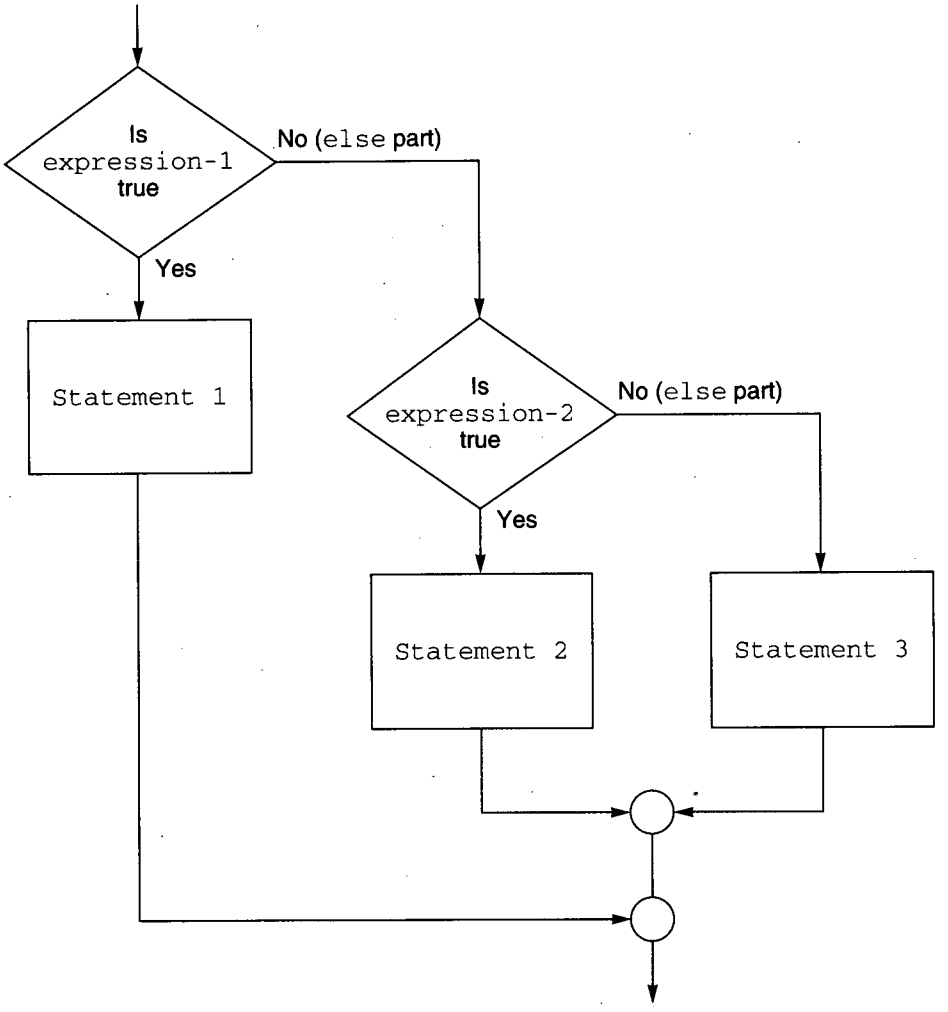


FIGURE 4–5b if-else Nested within else

As with all C programs, the indentation we have used is not required. Since white space is ignored in C, the above construction can be rewritten using the following arrangement:

```
if (expression_1)
  statement1;
else if (expression_2)
  statement2;
else
  statement3;
```

This form of a nested if statement is extremely useful in practice. It is formally referred to as an if-else chain. Each condition is evaluated in order, and if any condition is true the corresponding statement is executed and the remainder of the chain is terminated. The statement associated with the final else is only

executed if none of the previous conditions are satisfied. This serves as a default or catch-all case that is useful for detecting an impossible or error condition.

The chain can be continued indefinitely by repeatedly making the last statement another `if-else` statement. Thus, the general form of an `if-else` chain is:

```
if (expression_1)
  statement1;
else if (expression_2)
  statement2;
else if (expression_3)
  statement3;
      .
      .
      .
else if (expression_n)
  statement_n;
else
  last_statement;
```

Each condition is evaluated in the order it appears in the statement. For the first condition that is true, the corresponding statement is executed and the remainder of the statements in the chain are not executed. Thus, if `expression_1` is true, only `statement1` is executed; otherwise `expression_2` is tested. If `expression_2` is then true, only `statement2` is executed; otherwise `expression_3` is tested, and so on. The final `else` in the chain is optional, and `last_statement` is only executed if none of the previous expressions were true.

As a specific example, consider the following `if-else` chain:

```
if (marcode == 'M')
  printf("\nIndividual is married.");
else if (marcode == 'S')
  printf("\nIndividual is single.");
else if (marcode == 'D')
  printf("\nIndividual is divorced.");
else if (marcode == 'W')
  printf("\nIndividual is widowed.")
else
  printf("An invalid code was entered.");
```

Execution through this chain begins with the testing of the expression `marcode == 'M'`. If the value in `marcode` is an `M` the message `Individual is married` is displayed, no further expressions in the chain are evaluated, and execution resumes with the next statement immediately following the chain. If the value in `marcode` was not an `M`, the expression `marcode == 'S'` is tested, and so on, until a true condition is found. If none of the conditions in the chain are true, the message `An invalid code was entered` would be displayed. In all cases, execution resumes with whatever statement immediately follows the chain.

Program 4-5 uses this if-else chain within a complete program.

Program 4-5

```
#include <stdio.h>
main( )
{
  char marcode;

  printf("Enter a marital code: ");
  scanf("%c", &marcode);

  if (marcode == 'M')
    printf("\nIndividual is married.");
  else if (marcode == 'S')
    printf("\nIndividual is single.");
  else if (marcode == 'D')
    printf("\nIndividual is divorced.");
  else if (marcode == 'W')
    printf("\nIndividual is widowed.");
  else
    printf("\nAn invalid code was entered.");

  printf("\nThanks for participating in the survey");
}
```

In reviewing Program 4-5 note that the message Thanks for participating in the survey is always printed. This is the statement immediately after the if-else chain to which execution is transferred once the chain completes its execution. Which message is printed within the if-else chain depends on the value entered into marcode.

As with all C statements, each individual statement within the if-else chain can be replaced by a compound statement bounded by the braces { and }. As an example requiring a compound statement, consider an if-else statement that calculates a taxi-ride fare. In most cities taxis charge a set amount for the first mile, followed by an additional charge for each additional mile. Assuming that the initial mileage charge is $1.90 and any additional distance is charged at the rate of $1.25 per mile, the following if-else statement, using compound statements, can be used to determine and display the cost of a trip:

```
if (distance > 1.0)
{
  fare = 1.90 + (distance - 1.0) * 1.25;
  printf("The distance is greater than one mile");
  printf("\n and the total fare is $%5.2f.", fare);
}
else
```

```
{
  printf("The distance was under a mile and the standard");
  printf("\n fare of $1.90 is charged.");
}
```

As a final example illustrating the `if-else` chain, let us calculate the monthly income of a computer salesperson using the following commission schedule:

Monthly Sales	Income
greater than or equal to $50,000	$375 plus 16% of sales
less than $50,000 but greater than or equal to $40,000	$350 plus 14% of sales
less than $40,000 but greater than or equal to $30,000	$325 plus 12% of sales
less than $30,000 but greater than or equal to $20,000	$300 plus 9% of sales
less than $20,000 but greater than or equal to $10,000	$250 plus 5% of sales
less than $10,000	$200 plus 3% of sales

The following `if-else` chain can be used to determine the correct monthly income, where the variable `mon_sales` is used to store the salesperson's current monthly sales:

```
if (mon_sales >= 50000.00)
  income = 375.00 + .16 * mon_sales;
else if (mon_sales >= 40000.00)
  income = 350.00 + .14 * mon_sales;
else if (mon_sales >= 30000.00)
  income = 325.00 + .12 * mon_sales;
else if (mon_sales >= 20000.00)
  income = 300.00 + .09 * mon_sales;
else if (mon_sales >= 10000.00)
  income = 250.00 + .05 * mon_sales;
else
  income = 200.000 + .03 * mon_sales;
```

Notice that this example makes use of the fact that the chain is stopped once a true condition is found. This is accomplished by checking for the highest monthly sales first. If the salesperson's monthly sales is less than $50,000, the `if-else`

chain continues checking for the next highest sales amount until the correct category is obtained.

Program 4-6 uses this `if-else` chain to calculate and display the income corresponding to the value of monthly sales input to the `scanf( )` function.

Program 4-6

```
#include a<stdio.h>
main( )
{
  float mon_sales, income;

  printf("Enter the value of monthly sales: ");
  scanf("%f", &mon_sales);

  if (mon_sales >= 50000.00)
    income = 375.00 + .16 * mon_sales;
  else if (mon_sales >= 40000.00)
    income = 350.00 + .14 * mon_sales;
  else if (mon_sales >= 30000.00)
    income = 325.00 + .12 * mon_sales;
  else if (mon_sales >= 20000.00)
    income = 300.00 + .09 * mon_sales;
  else if (mon_sales >= 10000.00)
    income = 250.00 + .05 * mon_sales;
  else
    income = 200.00 + .03 * mon_sales;

  printf("The income is $%7.2f",income);
}
```

A sample run using Program 4-6 is illustrated below.

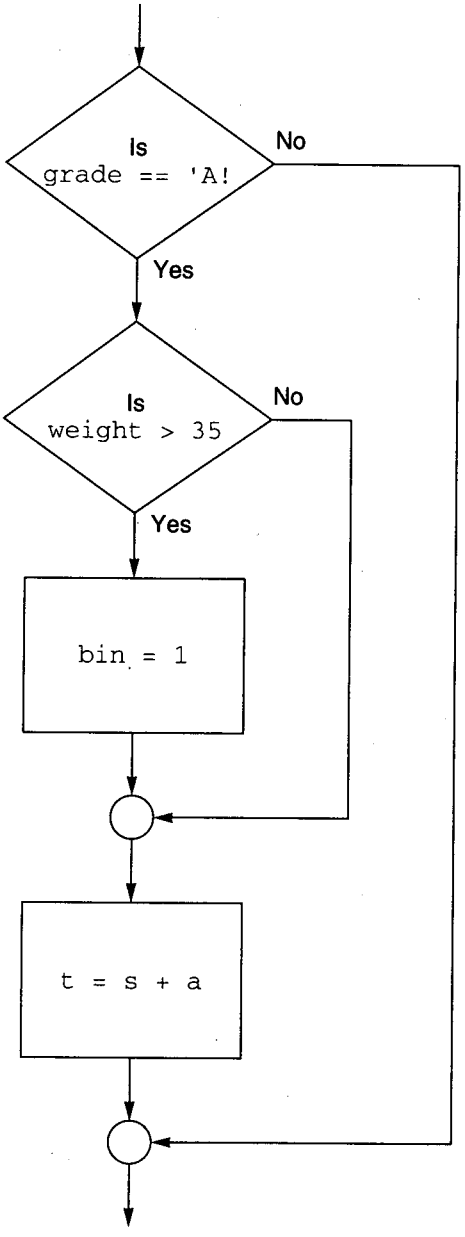
```
Enter the value of monthly sales: 36243.89
The income is $4674.27
```

Exercises 4.3

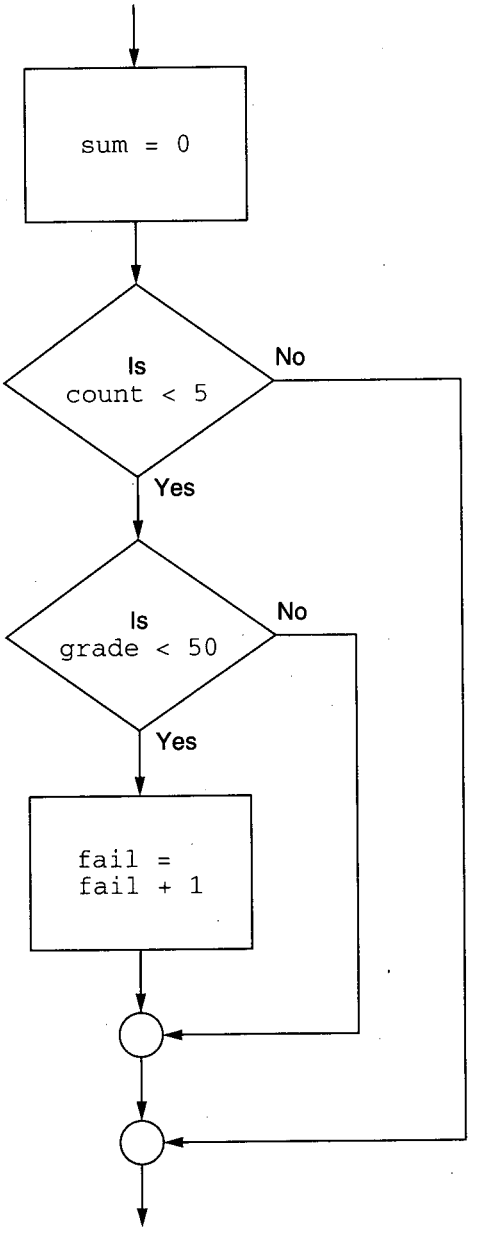
1. Modify Program 4-5 to accept both lower and uppercase letters as marriage codes. For example, if a user enters either an `m` or an `M`, the program should display the message `Individual is married.`

2. Write nested `if` statements corresponding to the conditions illustrated in each of the following flowcharts.

a.



b.



3. An angle is considered acute if it is less than 90 degrees, obtuse if it is greater than 90 degrees, and a right angle if it is equal to 90 degrees. Using this information write a C program that accepts an angle in degrees and displays the type of angle corresponding to the degrees entered.

4. The grade level of undergraduate college students is typically determined according to the following schedule:

Number of Credits Completed	Grade Level
less than 32	Freshman
32 to 63	Sophomore
64 to 95	Junior
96 or more	Senior

Using this information, write a C program that accepts the number of credits a student has completed, determines the student's grade level, and displays the grade level.

5. A student's letter grade is calculated according to the following schedule:

Numerical Grade	Letter Grade
greater than or equal to 90	A
less than 90 but greater than or equal to 80	B
less than 80 but greater than or equal to 70	C
less than 70 but greater than or equal to 60	D
less than 60	F

Using this information, write a C program that accepts a student's numerical grade, converts the numerical grade to an equivalent letter grade, and displays the letter grade.

6. The interest rate used on funds deposited in a bank is determined by the amount of time the money is left on deposit. For a particular bank, the following schedule is used:

Time on Deposit	Interest Rate
greater than or equal to 5 years	095
less than 5 years but greater than or equal to 4 years	09
less than 4 years but greater than or equal to 3 years	085
less than 3 years but greater than or equal to 2 years	075
less than 2 years but greater than or equal to 1 year	065
less than 1 year	058

Using this information, write a C program that accepts the time that funds are left on deposit and displays the interest rate corresponding to the time entered.

7. Write a C program that accepts a number followed by one space and then a letter. If the letter following the number is `f`, the program is to treat the number entered as a temperature in degrees Fahrenheit, convert the number to the equivalent degrees Celsius, and print a suitable display message. If the letter following the number is `c`, the program is to treat the number entered as a temperature in Celsius, convert the number to the equivalent degrees Fahrenheit, and print a suitable display message. If the letter is neither `f` nor `c` the program is to print a message that the data entered is incorrect and terminate. Use an `if-else` chain in your program and make use of the conversion formulas:

Celsius = (5.0 / 9.0) * (Fahrenheit − 32.0)
Fahrenheit = (9.0 / 5.0) * Celsius + 32.0

8. Using the commission schedule from Program 4-6, the following program calculates monthly income:

```
#include <stdio.h>
main( )
{
  float mon_sales, income;

  printf("Enter the value of monthly sales: ");
  scanf("%f",mon_sales);

  if (mon_sales >= 50000.00)
    income = 375.00 + .16 * mon_sales;
  if (mon_sales >= 40000.00 && mon_sales < 50000.00)
    income = 350.00 + .14 * mon_sales;
  if (mon_sales >= 30000.00 && mon_sales < 40000.00)
    income = 325.00 + .12 * mon_sales;
  if (mon_sales >= 20000.00 && mon_sales < 30000.00)
    income = 300.00 + .09 * mon_sales;
  if (mon_sales >= 10000.00 && mon_sales < 20000.00)
    income = 250.00 + .05 * mon_sales;
  if (mon_sales < 10000.00)
    income = 200.00 + .03 * mon_sales;

  printf("The income is $%7.2f",income);
}
```

a. Will this program produce the same output as Program 4-6?
b. Which program is better and why?

9. The following program was written to produce the same result as Program 4-6:

```
#include <stdio.h>
main( )
{
  float mon_sales, income;

  printf("Enter the value of monthly sales: ");
  scanf("%f",mon_sales);

  if (mon_sales < 10000.00)
    income = 200.00 + .03 * mon_sales;
  else if (mon_sales >= 10000.00)
    income = 250.00 + .05 * mon_sales;
  else if (mon_sales >= 20000.00)
    income = 300.00 + .09 * mon_sales;
  else if (mon_sales >= 30000.00)
    income = 325.00 + .12 * mon_sales;
  else if (mon_sales >= 40000.00)
    income = 350.00 + .14 * mon_sales;
  else if (mon_sales >= 50000.00)
    income = 375.00 + .16 * mon_sales;

  printf("The income is $%7.2f",income);
}
```

a. Will this program run?
b. What does this program do?
c. For what values of monthly sales does this program calculate the correct income?

4.4 The `switch` Statement

The `if-else` chain is used in programming applications where one set of instructions must be selected from many possible alternatives. The `switch` statement provides an alternative to the `if-else` chain for cases that compare the value of an integer expression to a specific value. The general form of a `switch` statement is:

```
switch(expression)
{                          /* start of compound statement */
  case value_1:               <------terminated with a colon
    statement1;
    statement2;
        .
        .
    break;
  case value_2:               <------terminated with a colon
    statementm;
    statementn;
        .
        .
    break;
        .
        .
  case value_n:               <------terminated with a colon
    statementw;
    statementx;
        .
        .
    break;
  default:                    <------terminated with a colon
    statementaa;
    statementbb;
        .
        .
}                      /* end of switch and compound statement */
```

The `switch` statement uses four new keywords: `switch`, `case`, `default`, and `break`. Let's see what each of these words does.

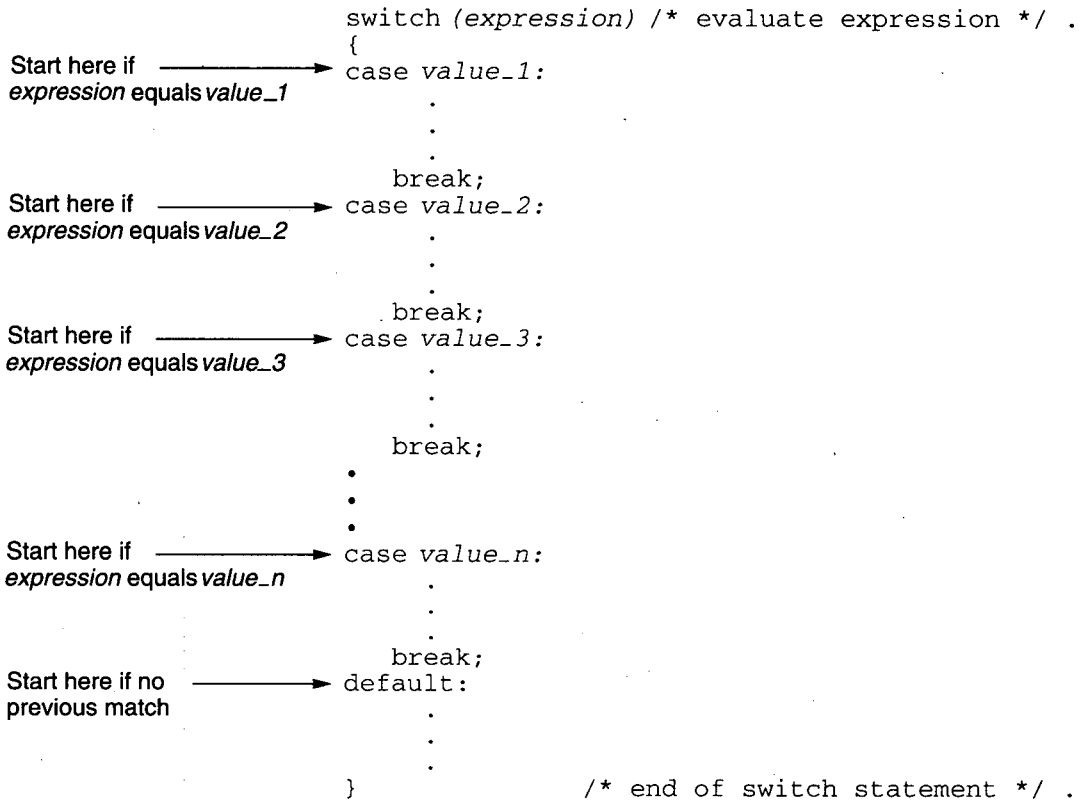
The keyword `switch` identifies the start of the `switch` statement. The expression in parentheses following this word is evaluated and the result of the expression compared to various alternative values contained within the compound statement. The expression in the `switch` statement must evaluate to an integer result or a compilation error results.

Internal to the switch statement, the keyword case is used to identify or label individual values that are compared to the value of the switch expression. The switch expression's value is compared to each of these case values in the order that these values are listed until a match is found. When a match occurs, execution begins with the statement immediately following the match. Thus, as illustrated in Figure 4–6, the value of the expression determines where in the switch statement execution actually begins.

Any number of case labels may be contained within a switch statement, in any order. If the value of the expression does not match any of the case values, however, no statement is executed unless the keyword default is encountered. The word default is optional and operates the same as the last else in an if-else chain. If the value of the expression does not match any of the case values, program execution begins with the statement following the word default.

Once an entry point has been located by the switch statement, all further case evaluations are ignored and execution continues through the end of the compound statement unless a break statement is encountered. This is the reason for the break statement, which identifies the end of a particular case and causes an immediate exit from the switch statement. Thus, just as the word case identifies possible starting points in the compound statement, the break statement determines terminating points. If the break statements are omitted, all cases following the matching case value, including the default case, are executed.

FIGURE 4–6 The Expression Determines an Entry Point



When we write a switch statement, we can use multiple case values to refer to the same set of statements; the default label is optional. For example, consider the following:

```
switch(number)
{
  case 1:
    printf("Have a Good Morning");
    break;
  case 2:
    printf("Have a Happy Day");
    break;
  case 3: case 4: case 5:
    printf("Have a Nice Evening");
}
```

If the value stored in the variable number is 1, the message Have a Good Morning is displayed. Similarly, if the value of number is 2, the second message is displayed. Finally, if the value of number is 3 or 4 or 5, the last message is displayed. Since the statement to be executed for these last three cases is the same, the cases for these values can be "stacked together" as shown in the example. Also, since there is no default, no message is printed if the value of number is not one of the listed case values. Although it is good programming practice to list case values in increasing order, this is not required by the switch statement. A switch statement may have any number of case values, in order; only the values being tested for need be listed.

Program 4-7 (see next page) uses a switch statement to select the arithmetic operation (addition, multiplication, or division) to be performed on two numbers depending on the value of the variable opselect.

Program 4-7 was run twice. The resulting display clearly identifies the case selected. The results are:

```
Please type in two numbers: 12 3
Enter a select code:
          1 for addition
          2 for multiplication
          3 for division : 2
The product of the numbers entered is 36.000
```

and

```
Please type in two numbers: 12 3
Enter a select code:
          1 for addition
          2 for multiplication
          3 for division : 3
The first number divided by the second is 4.000
```

In reviewing Program 4-7 notice the break statement in the last case. Although this break is not necessary, it is a good practice to terminate the last

Program 4-7

```
#include <stdio.h>
main( )
{
  int opselect;
  double fnum, snum;

  printf("Please type in two numbers: ");
  scanf("%lf %lf", &fnum, &snum);
  printf("Enter a select code: ");
  printf("\n         1 for addition");
  printf("\n         2 for multiplication");
  printf("\n         3 for division : ");
  scanf("%d", &opselect);

  switch(opselect)
  {
    case 1:
      printf("The sum of the numbers entered is %6.31f", fnum+snum);
      break;
    case 2:
      printf("The product of the numbers entered is %6.31f", fnum*snum);
      break;
    case 3:
      printf("The first number divided by the second is %6.31f", fnum/snum);
      break;   /* this break is optional */
  }   /* end of switch */

}.    /* end of main ( ) */
```

case in a switch statement with a break. This prevents a possible program error later, if an additional case is subsequently added to the switch statement. With the addition of a new case, the break between cases becomes necessary; having the break in place ensures you will not forget to include it at the time of the modification.

Since character data types are always converted to integers in an expression, a switch statement can also be used to "switch" based on the value of a character expression. For example, assuming that choice is a character variable, the following switch statement is valid:

```
switch(choice)
{
  case 'a': case 'e': case 'i': case 'o': case 'u':
    printf("\nThe character in choice in the vowel");
    break
  default:
  printf("\nThe character in choice is not a vowel");
}          /* end of switch statement */
```

Exercises 4.4

1. Rewrite the following `if-else` chain using a `switch` statement:

```
if (let_grad == 'A')
  printf("The numerical grade is between 90 and 100");
else if (let_grad == 'B')
  printf("The numerical grade is between 80 and 89.9");
else if (let_grad == 'C')
  printf("The numerical grade is between 70 and 79.9");
else if (let_grad == 'D');
  printf("How are you going to explain this one");
else
{
  printf("Of course I had nothing to do with my grade.");
  printf("\nThe professor was really off the wall.");
}
```

2. Rewrite the following `if-else` chain using a `switch` statement:

```
if (res_typ == 1)
{
  in_data( );
  check( );
}
else if (res_typ == 2)
{
  capacity( )
  devtype( )
}
  else if (res_typ == 3)
{
  volume( );
  mass( );
}
  else if (res_typ == 4)
{
  area( );
  weight( );
}
else if (res_typ == 5)
{
  files( );
  save( );
}
else if (res_type == 6)
{
  retrieve( );
  screen( );
}
```

3. Each disk drive in a shipment of these devices is stamped with a code from 1 through 4, which indicates a drive of the following type:

Code	Disk Drive Type
1	360 kilobyte drive (5 1/2 inch)
2	1.2 megabyte drive (5 1/2 inch)
3	722 kilobyte drive (3 1/4 inch)
4	1.4 megabyte drive (3 1/4 inch)

Write a C program that accepts the code number as an input and, based on the value entered, displays the correct disk drive type.

4. Rewrite Program 4-5 in Section 4-3 using a `switch` statement.

5. Determine why the `if-else` chain in Program 4-6 cannot be replaced with a `switch` statement.

6. Repeat Exercise 3 in Section 4-3 using a `switch` statement instead of an `if-else` chain.

7. Rewrite Program 4-7 using a character variable for the select code. (*Hint:* Review Section 3.3 if your program does not operate as you think it should.)

4.5 Applications

Two major uses of C's if statements are to select appropriate processing paths and to filter undesirable data from being processed at all. In this section examples of both uses are presorted.

Application 1: Solving Quadratic Equations

A *quadratic* equation is an equation that has the form $ax^2 + bx + c = 0$ or that can be algebraically manipulated into this form. In this equation x is the unknown variable and a, b, and c are known constants. Although the constants b and c can be any numbers, including zero, the value of the constant a cannot be zero (if a is zero, the equation would become a *linear* equation in x). Examples of quadratic equations are:

$$5x^2 + 6x + 2 = 0$$
$$x^2 - 7x + 20 = 0$$
$$34x^2 + 16 = 0$$

In the first equation $a = 5$, $b = 6$, and $c = 2$; in the second equation $a = 1$, $b = -7$, and $c = 20$; and in the third equation $a = 34$, $b = 0$, and $c = 16$.

The real roots of a quadratic equation can be calculated using the quadratic formula as:

$$root\ 1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

and

$$root\ 2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

A C program that solves for the two roots of a quadratic equation, without any data validation statements, would use the user-entered values of *a*, *b*, and *c* to directly calculate a value for each of the roots. However, if the user entered a value of 0 for *a*, the division by 2*a* would result in an error when the program is run. Another error occurs when the value of the term $b^2 - 4ac$, which is called the *discriminant*, is negative, because the square root of a negative number cannot be taken. The complete logic for correctly determining the roots of a quadratic equation, including the steps necessary to determine that a valid quadratic equation is being processed, is illustrated in Figure 4–7 (see page 180). The pseudocode corresponding to this figure is:

```
display a program purpose message
display a prompt for the coefficients
accept user input values for a, b, and c
if a = 0 and b = 0 then
  display a message saying that the equation
  is degenerate (has no solution)
else if a = zero then
  calculate the single root equal to -c/b
  display the single root
else
  calculate the discriminant
  if the discriminant > 0 then
    solve for both roots using the quadratic formula
    display the two roots
  else if the discriminant < 0 then
    display a message that there are no real roots
  else
    calculate the repeated root equal to -b/(2a)
    display the repeated root
  endif
endif
```

Notice in both the flowchart and the equivalent pseudocode that we have used nested `if` statements. The outer `if` statement is used to validate the entered coefficients and determine that we have a valid quadratic equation. The inner `if` statement is then used to determine if the equation has two real roots (`discriminant > 0`), two imaginary roots (`discriminant <0`), or repeated roots (`discriminant == 0`). The equivalent C code for this problem is listed in Program 4-8.

Following are two sample runs of Program 4-8:

```
This program calculates the roots of a
   quadratic equation of the form
          ax^2 + bx + c = 0

Please enter values for a, b, and c: 1 2 -35

The two real roots are 5.000000 and -7.000000
```

and

Program 4-8

```
/* this program solves for the roots of a quadratic equation */
#include <math.h>
#include <stdio.h>
main( )
{
  double a, b, c, disc, root1, root2;

  printf("This program calculates the roots of a");
  printf("\n  quadratic equation of the form");
  printf("\n           2");
  printf("\n         ax + bx + c = 0");
  printf("\n\nPlease enter values for a, b, and c: ");
  scanf("%lf %lf %lf", &a, &b, &c);
  if ( a == 0.0 && b == 0.0)
    printf("\nThe equation is degenerate and has no roots");
  else if (a == 0.0)
    printf("\nThe equation has the single root x = %lf", -c/b);
  else
  {
    disc = pow(b,2.0) - 4 * a * c;  /* calculate discriminant */
    if (disc > 0.0)
    {
      disc = sqrt(disc);
      root1 = (-b + disc) / (2 * a);
      root2 = (-b - disc) / (2 * a);
      printf("\nThe two real roots are %lf and %lf", root1, root2);
    }
    else if (disc < 0.0)
      printf("\nBoth roots are imaginary");
    else
      printf("\nBoth roots are equal to %lf", -b/(2 * a));
  }
}
```

```
This program calculates the roots of a
   quadratic equation of the form
          ax^2 + bx + c = 0

Please enter values for a, b, and c: 0 0 16

The equation is degenerate and has no roots
```

The first run solves the quadratic equation $x^2 + 2x - 35 = 0$, which has the real roots $x = 5$ and $x = -7$. The input data for the second run results in the equation $0x^2 + 0x + 16 = 0$. As this degenerates into the mathematical impossibility $16 = 0$, the program correctly identifies this as a degenerate equation.

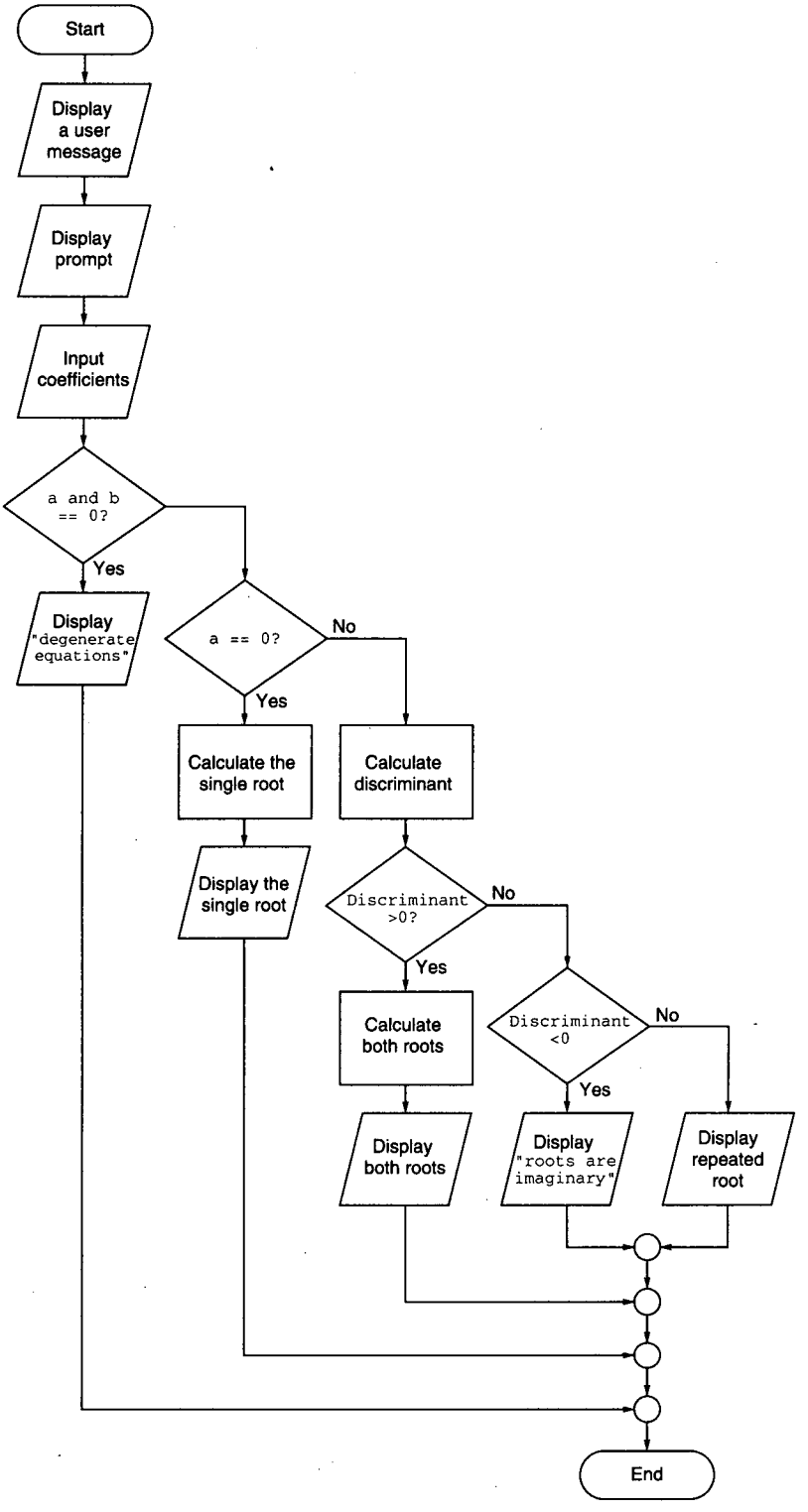


FIGURE 4–7 Program Logic Flowchart

Application 2: Data Validation

An important use of C's `if` statements is to validate data by checking for clearly invalid cases. For example, a date such as 5/33/86 contains an obviously invalid day. Similarly, the division of any number by zero within a program, such as 14/0, should not be allowed. Both of these examples illustrate the need for a technique called *defensive programming,* in which the program includes code to check for improper data before attempting to process it further. The defensive programming technique of checking user-input data for erroneous or unreasonable data is referred to as *input data validation.*

Consider this case: We are to write a C program to calculate the square root and the reciprocal of a user-entered number. Since the square root of a negative number does not exist as a real number and the reciprocal of zero cannot be taken, our program will contain input data validation statements to screen the user-input data to avoid these two cases.

The flowchart describing the processing required for our program is shown in Figure 4–8. The pseudocode corresponding to this flowchart logic is:

```
display a program purpose message
display a prompt for a number
accept a user input number
if the number is negative then
    print a message that the square root
    cannot be taken
else
    calculate and display the square root
endif
if the number is zero then
     print a message that the reciprocal
    cannot be taken
else
    calculate and display the reciprocal
endif
```

The C code corresponding to Figure 4–8 is listed in Program 4-9.

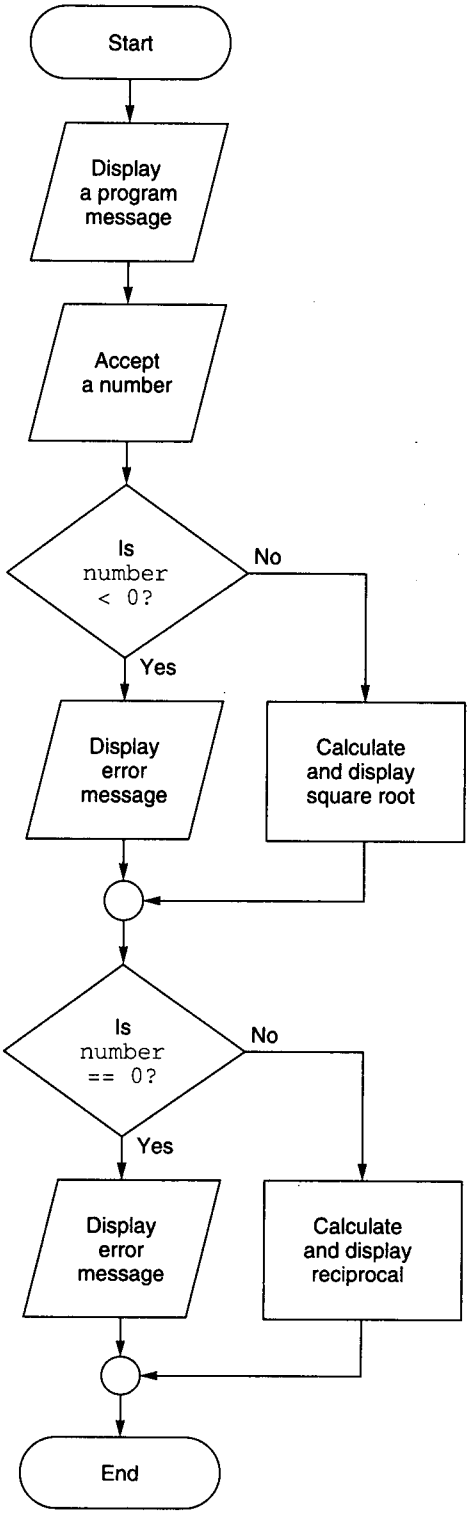


FIGURE 4–8 Program Logic Flowchart

Program 4-9

```
#include <math.h>
#include <stdio.h>
main( )
{
  double usenum;

  printf("This program calculates the square root and");
  printf("\nreciprocal (1/number) of a number");
  printf("\n\nPlease enter a number: ");
  scanf("%lf", &usenum);
  if (usenum < 0.0)
    printf("\nThe square root of a negative number does not exist");
  else
    printf("\nThe square root of %lf is %lf", usenum, sqrt(usenum));
  if (usenum == 0.0)
    printf("\nThe reciprocal of zero does not exits");
  else
  printf("\nThe reciprocal of %lf is %lf", usenum, 1/usenum);
}
```

Program 4-9 is a rather straightforward program containing two separate (nonnested) `if` statements. The first `if` statement checks for a negative input number; if the number is negative a message indicating that the square root of a negative number cannot be taken is displayed, else the square root is taken. The second `if` statement checks if the entered number is zero; if it is, a message indicating that the reciprocal of zero cannot be taken is displayed, else the reciprocal is taken. Following are two sample runs of Program 4-9:

```
This program calculates the square root and
reciprocal (1/number) of a number

Please enter a number: 5

The square root of 5.000000 is 2.236068
The reciprocal of 5.000000 is 0.200000
```

and

```
This program calculates the square root and
reciprocal (1/number) of a number

Please enter a number: -6

The square root of a negative number does not exist
The reciprocal of -6.000000 is -0.166667
```

Additional Exercises for Chapter 4

1 *a.* Write a program that accepts two real numbers and a select code a from user . If the entered select code is 1, have the program add the two previously entered numbers and display the result; if the select code is 2, the numbers should be multiplied, and if the select code is 3, the first number should be divided by the second number.
b. Determine what the program written in Exercise 1a does when the entered numbers are 3 and 0, and the select code is 3.
c. Modify the program written in Exercise 1a so that division by 0 is not allowed and that an appropriate message is displayed when such a division is attempted.

2 *a.* Write a program to display the following two prompts:

```
Enter a month (use a 1 for Jan, etc.):
Enter a day of the month:
```

Have your program accept and store a number in the variable `month` in response to the first prompt, and accept and store a number in the variable `DAY` in response to the second prompt. If the month entered is not between 1 and 12 inclusive, print a message informing the user that an invalid month has been entered. If the day entered is not between 1 and 31, print a message informing the user that an invalid day has been entered.
b. What will your program do if the user types a number with a decimal point for the month? How can you ensure that your `if` statements check for an integer number?
c. In a non-leap year February has 28 days, the months January, March, May, July, August, October, and December have 31 days, and all other months have 30 days. Using this information modify the program written in Exercise 2a to display a message when an invalid day is entered for a user-enentered month. For this program ignore leap years.

3 *a.* The quadrant that a line drawn from the origin resides in is determined by the angle that the line makes with the positive *X* axis as follows:

Angle from the Positive X Axis	Quadrant
Between 0 and 90 degrees	I
Between 90 and 180 degrees	II
Between 180 and 270 degrees	III
Between 270 and 360 degrees	IV

Using this information, write a C program that accepts the angle of the line as user input and determines and displays the quadrant appropriate to the input data. (*Note:* If the angle is exactly 0, 90, 180, or 270 degrees the corresponding line does not reside in any quadrant but lies on an axis.)
b. Modify the program written for Exercise 3a so that a message is displayed that identifies an angle of zero degrees as the positive *X* axis, an angle of 90 degrees as the positive *Y* axis, an angle of 180 degrees as the negative *X* axis, and an angle of 270 degrees as the negative *Y* axis.

4 *a.* All years that are evenly divisible by 400 or are evenly divisible by four and not evenly divisible by 100 are leap years. For example, since 1600 is evenly divisible by 400, the year 1600 was a leap year. Similarly, since 1988 is evenly divisible by four but not by 100, the year 1988 was also a leap year. Using this information, write a C program that accepts the year as a user input, determines if the year is a leap year, and displays an appropriate message that tells the user if the entered year is or is not a leap year.

b. Using the code written in Exercise 4a redo Exercise 2c to take leap years into account.

5. Based on an automobile's model year and weight the state of New Jersey determines the car's weight class and registration fee using the following schedule:

Model Year	Weight	Weight Class	Registration Fee
1970 or earlier	less than 2,700 lbs	1	$16.50
	2,700 to 3,800 lbs	2	25.50
	more than 3,800 lbs	3	46.50
1971 to 1979	less than 2,700 lbs	4	27.00
	2,700 to 3,800 lbs	5	30.50
	more than 3,800 lbs	6	52.50
1980 or later	less than 3,500 lbs	7	19.50
	3,500 or more lbs	8	52.50

Using this information write a C program that accepts the year and weight of an automobile and determines and displays the weight class and registration fee for the car.

6. Modify Program 4-8 so that the imaginary roots are calculated and displayed when the discriminant is negative. For this case the two roots of the equation are

$$x_1 = \frac{-b}{2a} + \frac{sqrt\left[-\left(b^2 - 4ac\right)\right]}{2a} i$$

and

$$x_2 = \frac{-b}{2a} + \frac{sqrt\left[-\left(b^2 - 4ac\right)\right]}{2a} i$$

where i is the imaginary number symbol for the square root of -1. (*Hint:* Calculate the real and imaginary parts of each root separately.)

7. In the game of blackjack the cards 2 through 10 are counted at their face values regardless of suit, all face cards (jack, queen, and king) are counted as 10, and an ace is counted as either a 1 or an 11, depending on the total count of all the cards in a player's hand. The ace is counted as 11 only if the resulting total value of all cards in a player's hand does not exceed 21, else it is counted as a 1. Using this information write a C program that accepts three card values as inputs (a 1 corresponding to an ace, a 2 corresponding to a 2, and so on), calculates the total value of the hand appropriately, and displays the value of the three cards with a printed message.

4.6 Common Programming Errors

There are three programming errors common to C's selection statements.

1. Using the assignment operator, =, in place of the relational operator, ==. This can cause an enormous amount of frustration because any expression can be tested by an `if-else` statement. For example, the statement

```
if (opselect = 2)
   printf("Happy Birthday");
else
   printf("Good Day");
```

always results in the message `Happy Birthday` being printed, regardless of the initial value in the variable `opselect`. The reason for this is that the assignment expression `opselect = 2` has a value of 2, which is considered a true value in C. The correct expression to determine the value in `opselect` is `opselect == 2`.

2. The second error presents a typical debugging problem. Here an `if-else` statement appears to select an incorrect choice and the programmer mistakenly concentrates on the tested condition as the source of the problem. For example, assume that the following `if-else` statement is part of your program:

```
if (key == 'F')
{
  contemp = (5.0/9.0) * (intemp -32.0);
  printf("Conversion to Celsius was done");
}
else
{
  contemp = (9.0/5.0) * intemp + 32.0;
  printf("Conversion to Fahrenheit was done");
}
```

This statement will always display `Conversion to Celsius was done` when the variable `key` contains an `F`. Therefore, if this message is displayed when you believe `key` does not contain `F`, investigation of `key`'s value is called for. As a general rule, whenever a selection statement does not act as you think it should, make sure to test your assumptions about the values assigned to the tested variables by displaying their values. If an unanticipated value is displayed, you have at least isolated the source of the problem to the variables themselves, rather than the structure of the `if-else` statement. Then you will have to determine where and how the incorrect value was obtained.

3. The third error occurs when nested `if` statements are used and braces are not included to clearly indicate the desired structure. Without braces the compiler defaults to pairing `else`s with the closest unpaired `if`s, which sometimes destroys the original intent of the selection statement. To avoid this problem and to create code that is readily adaptable to change it is useful to write all `if-else` statements as compound statements in the form

```
if (expression)
{
   one or more statements in here

}
else
{
   one or more statements in here
}
```

By using this form, no matter how many statements are added later, the original integrity and intent of the `if` statement is maintained.

4.7 Chapter Summary

1. Relational expressions, which are also called *simple conditions*, are used to com-pare operands. If a relational expression is true, the value of the expression is the integer 1. If the relational expression is false, it has an integer value of 0. Relational expressions are created using the following relational operators:

Relational Operator	Meaning	Example
<	less than	age < 30
<	greater than	height > 6.2
<=	less than or equal to	taxable <= 20000
>=	greater than or equal to	temp >= 98.6
==	equal to	grade == 100
!=	not equal to	number != 250

2. More complex conditions can be constructed from relational expressions using C's logical operators, `&&` (AND), `||` (OR), and `!` (NOT).
3. `if-else` statements are used to select between two alternative statements based on the value of an expression. Although relational expressions are usually used for the tested expression, any valid expression can be used. In testing an expression, `if-else` statements interpret a nonzero value as true and a zero value as false. The general form of an `if-else` statement is:

```
if (expression)
  statement1;
else
  statement2;
```

   This is a two-way selection statement. If the expression has a nonzero value it is considered as true, and `statement1` is executed; otherwise `statement2` is executed.
4. `if-else` statements can contain other `if-else` statements. In the absence of braces, each `else` is associated with the closest unpaired `if`.
5. The `if-else` chain is a multiway selection statement having the general form:

```
if (expression_1)
  statement_1;
else if (expression_2)
  statement_2;
```

```
else if (expression_3)
  statement_3;
      .
      .
      .
else if (expression_m)
  statement_m;
else
  statement_n;
```

Each expression is evaluated in the order it appears in the chain. Once an expression is true (has a nonzero value), only the statement between that expression and the next `else if` or `else` is executed, and no further expressions are tested. The final `else` is optional, and the statement corresponding to the final `else` is only executed if none of the previous expressions were true.

6. A compound statement consists of any number of individual statements enclosed within the brace pair `{` and `}`. Compound statements are treated as a single unit and can be used anywhere a single statement is called for.
7. The `switch` statement is a multiway selection statement. The general form of a `switch` statement is:

```
switch(expression)
{                          /* start of compound statement */
  case value_1:               <———terminated with a colon
    statement1;
    statement2;
        .
        .
   break;
  case value_2:               <———terminated with a colon
    statementm;
    statementn
        .
        .
    break;
        .
        .

  case value_n:               <———terminated with a colon
    statementw;
    statementx;
        .
        .
    break;
  default:                    <———terminated with a colon
    statementaa;
    statementbb;
        .
        .
}                  /* end of switch and compound statement */
```

For this statement the value of an integer expression is compared to a number of integer or character constants or constant expressions. Program execution is transferred to the first matching `case` and continues through the end of the `switch` statement unless an optional `break` statement is encountered. `cases` in a `switch` statement can appear in any order and an optional `default` case can be included. The `default` case is executed if none of the other cases is matched.

4.8 Enrichment Study: A Closer Look at Errors, Testing, and Debugging

The ideal in programming is to efficiently produce readable, error–free programs that work correctly and can be modified or changed with a minimum of testing required for reverification. In this regard it is useful to know the different types of errors that can occur, when they are detected, and how they can be and corrected.

Compile–Time and Run–Time Errors

An error in a program can be detected either before a program is compiled, while the program is being compiled, while it is being run, after the program has been executed and the output is being examined, or not at all. Errors that are detected by the compiler are formally referred to as *compile-time* errors and errors that occur while the program is being run are formally referred to as *run-time* errors.

Although there is no formal name for errors that are detected either before a program is compiled or after a program is executed, there are methods for locating errors at these times. The method for detecting errors after a program has been executed is formally referred to as *program verification and testing*. The method for detecting errors before a program is compiled is called *desk checking*. Desk checking refers to the procedure of checking a program by hand at a desk or table for syntax and logic errors, which are described next.

Syntax and Logic Errors

Computer literature distinguishes between two primary types of errors, called syntax and logic errors, respectively. A *syntax error* is an error in the structure or spelling of a statement. For example, the statement

```
if ( a lt b
{
   printf("There are four syntax errors here\n")
   printf(" Can you find tem);
}
```

contains four syntax errors. These errors are:

1. The relational operator in the first line is incorrect and should be the symbol <.
2. The closing parenthesis is missing in line 1.
3. The third line is missing the terminating semicolon (;).
4. The string within parentheses in the fourth line is not terminated with quotes.

All of these errors will be detected by the compiler when the program is compiled. This is true of all syntax errors—since they violate the basic rules of C, if they are not discovered by desk checking, the compiler will detect them and display an error message indicating that a syntax error exists.[2] In some cases the error message is extremely clear and the error is obvious; in other cases it takes a little detective work to understand the error message displayed by the compiler. Since all syntax errors are detected at compile time, the terms compile-time and syntax errors are frequently used interchangeably. Strictly speaking, however, compile-time refers to when the error was detected and syntax refers to the type of error detected.

Note that the misspelling of the word `tem` in the second `printf( )` function call is not a syntax error. Although this spelling error will result in an undesirable output line being displayed, it is not a violation of C's syntactical rules. This spelling error is a rather simple example of a logic error.

Logic errors are characterized by erroneous, unexpected, or unintentional errors that are a direct result of some flaw in the program's logic. These errors, which are never caught by the compiler, may be detected either by desk checking, by program testing, by accident when a user obtains an obviously erroneous output, while the program is executing, or not at all. If the error is detected while the program is executing a run-time error occurs that results in an error message being generated and/or abnormal and premature program termination.

Since logic errors may not be detected by the compiler, they are always more difficult to detect than syntax errors. Logic errors typically cause one or more of the following to occur:

No output. This is caused either by an omission of a `printf( )` statement or a sequence of statements that inadvertentlybypasses a `printf( )` function call.

Unappealing or misaligned output. This is always caused by an error in a `printf( )` statement.

Incorrect numerical results. This is always caused by incorrect values assigned to the variables used in an expression,the use of an incorrect arithmetic expression, an omission of a statement, roundoff error, or the use of an improper sequence of statements. This type of error may also cause a run-time error.

See if you can detect the logic error in Program 4-10.
Following is a sample run of Program 4-10:

```
This program calculates the amount of money
in a bank account for an initial deposit
invested for n years at an interest rate r.

Enter the initial amount in the account: 1000.
Enter the interest rate (ex. 5 for 5%): 5

The final amount of money is $ 1000.00
```

[2] They may not, however, all the detected at the same time. Frequently, one syntax error "masks" another error and the second error is only detected after the first error is corrected.

Program 4-10

```
#include <math.h>
#include <stdio.h>
main( )  /* a compound interest program */
{
  float capital, amount, rate, nyrs;

  printf("This program calculates the amount of money\n");
  printf("in a bank account for an initial deposit\n");
  printf("invested for n years at an interest rate r.\n\n");
  printf("Enter the initial amount in the account: ");
  scanf("%f", &amount);
  printf("Enter the interest rate (ex. 5.0 for 5.0%): ");
  scanf("%f", &rate);
  capital = amount * pow( (1 + rate/100.), nyears);
  printf("\nThe final amount of money is $%8.2f", capital);
}
```

As indicated in the output, the final amount of money is identical to the initial amount input. Did you spot the error in Program 4-10 that produced this apparently erroneous output?

Unlike a misspelled output message, the error in Program 4-10 causes a mistake in a computation. Here the error is that the program does not initialize the variable `nyears` before this variable is used in the calculation of `capital`. When the assignment statement that calculates `capital` is executed the computer uses whatever value is stored in `nyears`. On those systems that initialize all variables to zero, the value 0 will be used for `nyears`. However, on those systems that do not initialize all variables to zero, whatever "garbage" value that happens to occupy the storage locations corresponding to the variable `nyears` will be used (the manuals supplied with your compiler will indicate which of these two actions your compiler takes). In either case an error is produced.

Although the logic error in this example program did not cause premature program termination, faulty or incomplete program logic can cause run-time errors. Examples of this type of logic error are attempts to divide by zero or take the square root of a negative number.

Testing and Debugging

In theory, a comprehensive set of test runs would reveal all logic errors and ensure that a program will work correctly for any and all combinations of input and computed data. In practice this requires checking all possible combinations of statement execution. Due to the time and effort required, this is an impossible goal except for extremely simple programs. Let us see why this is so. Consider Program 4-11.

Program 4-11

```
main( )
{
  int num;

  printf("Enter a number:");
  scanf("%d", &num);
  if (num == 5)
    printf("Bingo!");
  else
    printf("Bongo!");
}
```

Program 4-11 has two paths that can be traversed from when the program is run to when the program reaches its closing brace. The first path, which is executed when the input number is `5`, is in this sequence:

```
printf("Enter a number");
scanf("%d", &num);
printf("Bingo!");
```

The second path, which is executed whenever any number except `5` is input, includes this sequence of instructions:

```
printf("Enter a number");
scanf("%d", &num);
printf("Bongo!");
```

To test each possible path through Program 4-11 requires two runs of the program, with a judicious selection of test input data to ensure that both paths of the `if` statement are exercised. The addition of one more `if` statement in the program increases the number of possible execution paths by a factor of 2 and requires four (2^2) runs of the program for complete testing. Similarly, two additional `if` statements increase the number of paths by a factor of 4 and requires eight (2^3) runs for complete testing and three additional `if` statements would produce a program that required sixteen (2^4) test runs.

Now consider a modestly sized application program consisting of only 10 functions, each function containing five `if` statements. Assuming the functions are always called in the same sequence, there are 32 possible paths through each function (2 raised to the fifth power) and more than 1,000,000,000,000,000 (2 raised to the fiftieth power) possible paths through the complete program (all functions executed in sequence). The time needed to create individual test data to exercise each path and the actual computer run time required to check each path make the complete testing of such a program impossible to achieve.

The inability to fully test all combinations of statement execution sequences

has led to the programming proverb, "There is no error-free program." It has also led to the realization that any testing that is done should be well thought out to maximize the possibility of locating errors. An important corollary to this is the realization that although a single test can reveal the presence of an error, it does not verify the absence of one. The fact that one error is revealed by testing does not indicate that another error is not lurking somewhere else in the program; the fact that one test revealed no errors does not indicate that there are no errors.

Once an error is discovered, however, the programmer must locate where the error occurs, and then fix it. In computer jargon, a program error is referred to as a "bug" and the process of isolating, correcting, and verifying the correction is called "debugging."

Although there are no hard and fast rules for isolating the cause of an error, some useful techniques can be applied. The first is a preventive technique. Frequently many errors are simply introduced by the programmer in the rush to code and run a program before fully understanding what is required and how the result is to be achieved. A symptom of this haste to get a program entered into the computer is the lack of an outline of the proposed program (pseudocode or flowcharts) or a handwritten program itself. Many errors can be eliminated simply by desk checking a copy of the program before it is ever entered or compiled.

A second useful technique is to mimic the computer and execute each statement, by hand, as the computer would. This means writing down each variable as it is encountered in the program and listing the value that should be stored in the variable as each input and assignment statement is encountered. Doing this also sharpens your programming skills, because it requires that you fully understand what each statement in your program causes to happen. Such a check is called *program tracing*.

A third and very powerful debugging technique is to use one or more diagnostic `printf( )` function calls to display the values of selected variables. For example, consider Program 4-10 again. Since this program produced an incorrect value for `capital`, it is worthwhile placing a `printf( )` statement immediately before the assignment statement for `capital` to display the value of all variables used in the computation. If the displayed values are correct, then the problem is in the assignment statement; if the values are incorrect, we must determine where the incorrect values were actually obtained.

Another use of `printf( )` function calls in debugging is to immediately display the values of all input data. This technique is referred to as *echo printing*, and is useful in establishing that the computer is correctly receiving and interpreting the input data.

Finally, no discussion of debugging is complete without mentioning the primary ingredient needed for successful isolation and correction of errors: the attitude and spirit you bring to the task. Since you wrote the program your natural assumption is that it is correct or you would have changed it before it was compiled. It is extremely difficult to back away and honestly test and find errors in your own software. As a programmer you must constantly remind yourself that just because you *think* your program is correct does not make it so. Finding errors in your own programs is a sobering experience, but one that will help you become a master programmer. It can also be exciting and fun if approached as a detection problem with you as the master detective.

Repetition

Chapter Five

5.1 The `while` Statement
5.2 `scanf( )` Within a `while` Loop
5.3 The `for` Statement
5.4 `for` Loop Programming Applications
5.5 Nested Loops
5.6 The `do` Statement
5.7 Common Programming Errors
5.8 Chapter Summary

The programs examined so far have been useful in illustrating the correct structure of C programs and in introducing fundamental C input, output, assignment, and selection capabilities. By this time you should have gained enough experience to be comfortable with the concepts and mechanics of the C programming process. It is now time to move up a level in our knowledge and abilities.

The real power of most computer programs resides in their ability to repeat the same calculation or sequence of instructions many times over, each time using different data, without the necessity of rerunning the program for each new set of data values. In this chapter we explore the C statements that permit this. These statements are the `while`, `for`, and `do-while` statements.

5.1 The while Statement

The `while` statement is a general repetition statement that can be used in a variety of programming situations. The general form of the `while` statement is:

```
while (expression)
  statement;
```

The expression contained within the parentheses is evaluated in exactly the same manner as an expression contained in an `if-else` statement; the difference is how the expression is used. As we have seen, when the expression is true (has a nonzero value) in an `if-else` statement, the statement following the expression is executed once. In a `while` statement the statement following the expression is executed repeatedly as long as the expression retains a nonzero value. This naturally means that somewhere in the `while` statement there must be a statement that alters the value of the tested expression. As we will see, this is indeed the case. For now, however, considering just the expression and the statement following the parentheses, the process used by the computer in evaluating a `while` statement is:

1. test the expression
2. `if` the expression has a nonzero (true) value
   a. execute the statement following the parentheses
   b. go back to step 1

   `else`
   exit the `while` statement

Notice that step 2b forces program control to be transferred back to step 1. The transfer of control back to the start of a `while` statement in order to reevaluate the expression is an example of a program loop. The `while` statement literally loops back on itself to recheck the expression until it evaluates to zero (becomes false).

This looping process is illustrated in Figure 5–1. A diamond shape is used to show the two entry and two exit points required in the decision part of the `while` statement.

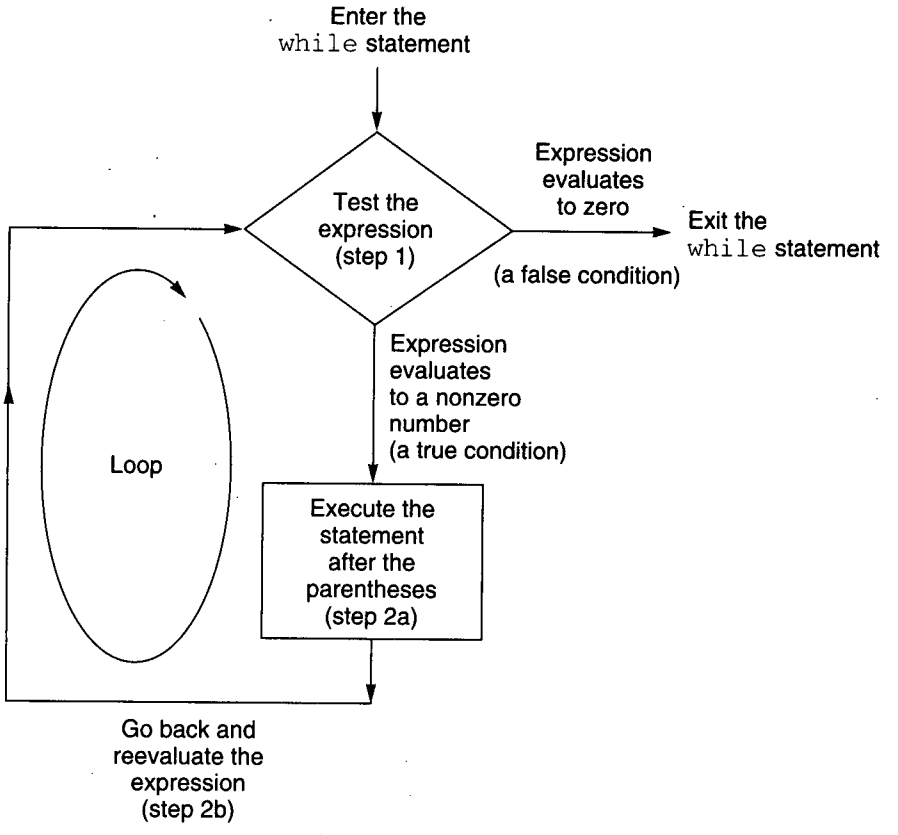


FIGURE 5–1 Anatomy of a `while` Loop

To make this a little more tangible, consider the relational expression `count <= 10` and the statement `printf("%d ",count);`. Using these, we can write the following valid `while` statement:

```
while (count <= 10)
  printf("%d ",count);
```

Although the above statement is valid, the alert reader will realize that we have created a situation in which the `printf( )` function either is called forever (or until we stop the program) or is not called at all. Let us see why this happens.

If `count` has a value less than or equal to `10` when the expression is first evaluated, a call to `printf( )` is made. The `while` statement then automatically loops back on itself and retests the expression. Since we have not changed the value stored in `count`, the expression is still true and another call to `printf( )` is made. This process continues forever, or until the program containing this statement is prematurely stopped by the user. However, if `count` starts with a value greater than `10`, the expression is false to begin with and the `printf( )` function call is never made.

How do we set an initial value in `count` to control what the `while` statement does the first time the expression is evaluated? The answer, of course, is to assign

values to each variable in the tested expression before the while statement is encountered. For example, the following sequence of instructions is valid:

```
count = 1;
while (count <= 10)
  printf("%d ",count);
```

Using this sequence of instructions, we have ensured that count starts with a value of 1. We could assign any value to count in the assignment statement—the important thing is to assign *some* value. In practice, the assigned value depends on the application.

We must still change the value of count so that we can finally exit the while statement. To do this requires an expression such as count = count + 1 to increment the value of count each time the while statement is executed. The fact that a while statement provides for the repetition of a single statement does not prevent us from including an additional statement to change the value of count. All we have to do is replace the single statement with a compound statement. For example:

```
count = 1;                  /* initialize count */
while (count <= 10)
{
  printf("%d ",count);
  count = count + 1;        /* increment count */
}
```

Note that, for clarity, we have placed each statement in the compound statement on a different line. This is consistent with the convention adopted for compound statements in the last chapter. Let us now analyze the above sequence of instructions.

The first assignment statement sets count equal to 1. The while statement is then entered and the expression is evaluated for the first time. Since the value of count is less than or equal to 10, the expression is true and the compound statement is executed. The first statement in the compound statement is a call to the printf( ) function to display the value of count. The next statement adds 1 to the value currently stored in count, making this value equal to 2. The while statement now loops back to retest the expression. Since count is still less than or equal to 10, the compound statement is again executed. This process continues until the value of count reaches 11. Program 5-1 illustrates these statements in an actual program.

The output for Program 5-1 is:

```
1  2  3  4  5  6  7  8  9  10
```

There is nothing special about the name count used in Program 5-1. Any valid integer variable could have been used.

Before we consider other examples of the while statement two comments concerning Program 5-1 are in order. First, the statement count = count + 1 can be replaced with any statement that changes the value of count. A statement such as count = count + 2, for example, would cause every second integer

Program 5-1

```
#include <stdio.h>
main( )
{
  int count;

  count = 1;                    /* initialize count */
  while (count <= 10)
  {
    printf("%d  ",count);
    count = count + 1;          /* add 1 to count */
  }
}
```

to be displayed. Second, it is the programmer's responsibility to ensure that `count` is changed in a way that ultimately leads to a normal exit from the `while`. For example, if we replace the expression `count = count + 1` with the expression `count = count - 1`, the value of `count` will never exceed `10` and an infinite loop will be created. An infinite loop is a loop that never ends. The computer will not reach out, touch you, and say, "Excuse me, you have created an infinite loop." It just keeps displaying numbers until you realize that the program is not working as you expected.

Now that you have some familiarity with the `while` statement, see if you can read and determine the output of Program 5-2.

Program 5-2

```
#include <stdio.h>
main( )
{
  int i;

  i = 10;
  while (i >= 1)
  {
    printf("%d  ",i);
    count = count - 1;
  }
}
```

The assignment statement in Program 5-2 initially sets the `int` variable `i` to `10`. The `while` statement then checks to see if the value of `i` is greater than or

equal to 1. While the expression is true, the value of i is displayed by the call to printf( ) and the value of i is decremented by 1. When i finally reaches zero, the expression is false and the program exits the while statement. Thus, the following display is obtained when Program 5-2 is run:

```
10  9  8  7  6  5  4  3  2  1
```

To illustrate the power of the while statement, consider the task of printing a table of numbers from 1 to 10 with their squares and cubes. This can be done with a simple while statement as illustrated by Program 5-3.

Program 5-3

```
#include <stdio.h>
main( )
{
  int num;

  printf("NUMBER     SQUARE     CUBE\n");
  printf("------     ------     ----\n");
  num = 1;
  while (num < 11)
  {
    printf("%3d         %3d       %4d\n", num, num*num,
num*num*num);
    num = num + 1;
  }
}
```

When Program 5-3 is run, the following display is produced:

```
NUMBER     SQUARE     CUBE
------     ------     ----
  1          1          1
  2          4          8
  3          9         27
  4         16         64
  5         25        125
  6         36        216
  7         49        343
  8         64        512
  9         81        729
 10        100       1000
```

Note that the expression used in Program 5-3 is num < 11. For the integer variable num, this expression is exactly equivalent to the expression num <= 10. The choice of which to use is entirely up to you.

If we want to use Program 5-3 to produce a table of 1000 numbers, all we do is change the expression in the `while` statement from `i < 11` to `i < 1001`. Changing the `11` to `1001` produces a table of 1000 lines—not bad for a simple five-line `while` statement.

All the program examples illustrating the `while` statement have checked for a fixed-count condition. Since any valid expression can be evaluated by a `while` statement we are not restricted to constructing such loops. For example, consider the task of producing a Celsius to Fahrenheit temperature conversion table. Assume that Fahrenheit temperatures corresponding to Celsius temperatures ranging from 5 to 50 degrees are to be displayed in increments of five degrees. The desired display can be obtained with this series of statements:

```
celsius = 5;      /* starting Celsius value */
while (celsius <= 50)
{
  fahren = (9.0/5.0) * celsius + 32.0;
  printf("%5d%12.2f",celsius, fahren);
  celsius = celsius + 5;
}
```

As before, the `while` statement consists of everything from the word `while` through the closing brace of the compound statement. Prior to entering the `while` loop we have made sure to assign a value to the operand being evaluated, and there is a statement to alter the value of `celsius` to ensure an exit from the `while` loop. Program 5-4 illustrates the use of this code in a complete program.

Program 5-4

```
#include <stdio.h>
main( ) /* program to convert Celsius to Fahrenheit */
{
  int celsius;
  float fahren;

  printf("DEGREES   DEGREES\n");
  printf("CELSIUS  FAHRENHEIT\n");
  printf("-------  ----------\n");

  celsius = 5;      /* starting Celsius value */
  while (celsius <= 50)
  {
    fahren = (9.0/5.0) * celsius + 32.0;
    printf("%5d%12.2f\n",celsius, fahren);
    celsius = celsius + 5;
  }
}
```

The display obtained when Program 5-4 is executed is:

```
DEGREES    DEGREES
CELSIUS   FAHRENHEIT
-------   ----------
   5          41.00
  10          50.00
  15          59.00
  20          68.00
  25          77.00
  30          86.00
  35          95.00
  40         104.00
  45         113.00
  50         122.00
```

Exercises 5.1

1. Rewrite Program 5-1 to print the numbers 2 to 10 in increments of two. The output of your program should be:

```
2  4  6  8  10
```

2. Rewrite Program 5-4 to produce a table that starts at a Celsius value of −10 and ends with a Celsius value of 60, in increments of 10 degrees.

3a. For the following program determine the total number of items displayed. Also determine the first and last numbers printed.

```
#include <stdio.h>
main( )
{
  int num = 0;

  while (num <= 20)
  {
    ++num;
    printf("%d  ",num);
  }
}
```

b. Enter and run the program from Exercise 3a on a computer to verify your answers to the exercise.

c. How would the output be affected if the two statements within the compound statement were reversed (that is, if the `printf( )` call were made before the `++num` statement)?

4. Write a C program that converts gallons to liters. The program should display gallons from 10 to 20 in 1-gallon increments and the corresponding liter equivalents. Use the relationship: `liters = 3.785 * gallons`.

5. Write a C program that converts feet to meters. The program should display feet from

3 to 30 in 3-foot increments and the corresponding meter equivalents. Use the relationship: `meters = feet / 3.28`.

6. A machine purchased for $28,000 is depreciated at a rate of $4,000 a year for 7 years. Write and run a C program that computes and displays a depreciation table for 7 years. The table should have the form:

Year	Depreciation	End-of-Year Value	Accumulated Depreciation
1	4000	24000	4000
2	4000	20000	8000
3	4000	16000	12000
4	4000	12000	16000
5	4000	8000	20000
6	4000	4000	24000
7	4000	0	28000

7. An automobile travels at an average speed of 55 miles per hour for 4 hours. Write a C program that displays the distance driven in miles that the car has traveled after .5, 1, 1.5, etc., hours until the end of the trip.

8a. An approximate conversion formula for converting Fahrenheit to Celsius temperatures is:

$$\text{Celsius} = (\text{Fahrenheit} - 30) / 2$$

Using this formula and starting with a Fahrenheit temperature of zero degrees, write a C program that determines when the approximate equivalent Celsius temperature differs from the exact equivalent value by more than 4 degrees. (*Hint:* Use a `while` loop that terminates when the difference between approximate and exact Celsius equivalents exceeds 4 degrees.)

b. Using the approximate Celsius conversion formula given in Exercise 8a, write a C program that produces a table of Fahrenheit temperatures, exact Celsius equivalent temperatures, approximate Celsius equivalent temperatures, and the difference between the correct and approximate equivalent Celsius values. The table should begin at zero degrees Fahrenheit, use 2-degree Fahrenheit increments, and terminate when the difference between exact and approximate values differs by more than 4 degrees.

9. The value of Euler's number, *e*, can be approximated using the formula

$$e = 1 + 1/1! + 1/2! + 1/3! + 1/4! + 1/5! + \ldots$$

Using this formula, write a C program that approximates the value of *e* using a `while` loop that terminates when the difference between two successive approximations differs by less than 1.0e-6.

10. The value of sin *x* can be approximated using the formula

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \ldots$$

Using this formula, determine how many terms are needed to approximate the value returned by the intrinsic `sin( )` function with an error less than 1.0e-6, when $x = 30$ degrees. (*Hints:* Use a `while` loop that terminates when the difference between the value returned by the intrinsic `sin( )` function and the approximation is less than 1e-6. Also note that *x* must first be converted to radian measure and that the alternating sign in the approximating series can be determined as $(-1)^{(n+1)}$ where *n* is the number of terms used in the approximation.)

5.2 scanf( ) Within a while Loop

Combining the scanf( ) function with the repetition capabilities of the while statement produces very adaptable and powerful programs. To understand the concept involved, consider Program 5-5, where a while statement is used to accept and then display four user-entered numbers, one at a time. Although it uses a very simple idea, the program highlights the flow of control concepts needed to produce more useful programs.

Program 5-5

```
#include <stdio.h>
main( )
{
  int count;
  float num;

  printf("\nThis program will ask you to enter some numbers.\n");
  count = 1;

  while (count <= 4)
  {
    printf("\nEnter a number: ");
    scanf("%f", &num);
    printf("The number entered is %f", num);
    count = count + 1;
  }
}
```

Following is a sample run of Program 5-5. The italicized items were input in response to the appropriate prompts.

```
This program will ask you to enter some numbers.

Enter a number: 26.2
The number entered is 26.200000
Enter a number: 5
The number entered is 5.000000
Enter a number: 103.456
The number entered is 103.456000
Enter a number: 1267.89
The number entered is 1267.890000
```

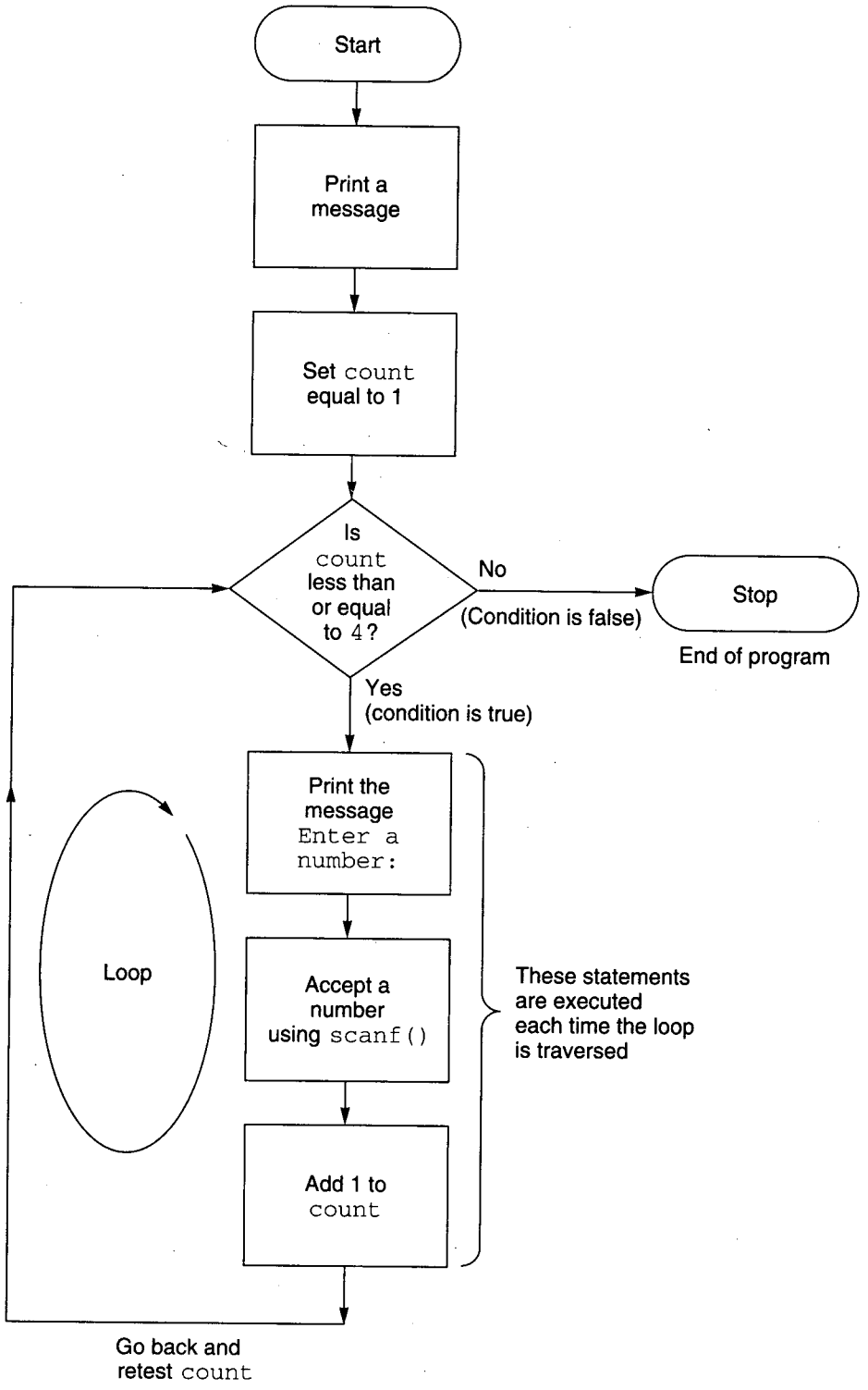


FIGURE 5–2 Flow of Control Diagram for Program 5-5

Let us review the program to clearly understand how the output was produced. The first message displayed is caused by execution of the first printf( ) function call. This call is outside and before the while statement, so it is executed once before any statement in the while loop.

Once the while loop is entered, the statements within the compound statement are executed while the tested condition is true. The first time through the compound statement, the message Enter a number: is displayed. The program then calls scanf( ), which forces the computer to wait for a number to be entered at the keyboard. Once a number is typed and the RETURN key is pressed, the call to printf( ) displaying the number is executed. The variable count is then incremented by one. This process continues until four passes through the loop have been made and the value of count is 5. Each pass causes the message Enter a number: to be displayed, causes one call to scanf( ) to be made, and causes the message The number entered is to be displayed. Figure 5–2 illustrates this flow of control.

Rather than simply displaying the entered numbers, Program 5-5 can be modified in order to use the entered data. For example, let us add the numbers entered and display the total. We must be careful in how we add the numbers, since the same variable, num, is used for each number entered. Because of this the entry of a new number in Program 5-5 automatically causes the previous number stored in num to be lost. Thus, each number entered must be added to the total before another number is entered. The required sequence is:

```
Enter a number
Add the number to the total
```

How do we add a single number to a total? A statement such as total = total + num does the job perfectly. This is the accumulating statement introduced in Section 2.3. After each number is entered, the accumulating statement adds the number into the total, as illustrated in Figure 5–3 (206). The complete flow of control required for adding the numbers is illustrated in Figure 5–4 (p. 208).

In reviewing Figure 5–4, observe that we have made a provision for initially setting the total to zero before the while loop is entered. If we were to clear the total inside the while loop, it would be set to zero each time the loop was executed and any value previously stored would be erased.

Program 5-6 (p. 207) incorporates the necessary modifications to Program 5–5 to total the numbers entered. As indicated in the flow diagram shown in Figure 5–4, the statement total = total + num; is placed immediately after the scanf( ) function call. Putting the accumulating statement at this point in the program ensures that the entered number is immediately "captured" into the total.

Let us review Program 5-6. The variable total was created to store the total of the numbers entered. Prior to entering the while statement the value of total is set to zero. This ensures that any previous value present in the storage location(s) assigned to the variable total is erased. Within the while loop the statement total = total + num; is used to add the value of the entered number into total. As each value is entered, it is added into the existing total to create a new total. Thus, total becomes a running subtotal of all the values entered. Only when all numbers are entered does total contain the final sum of

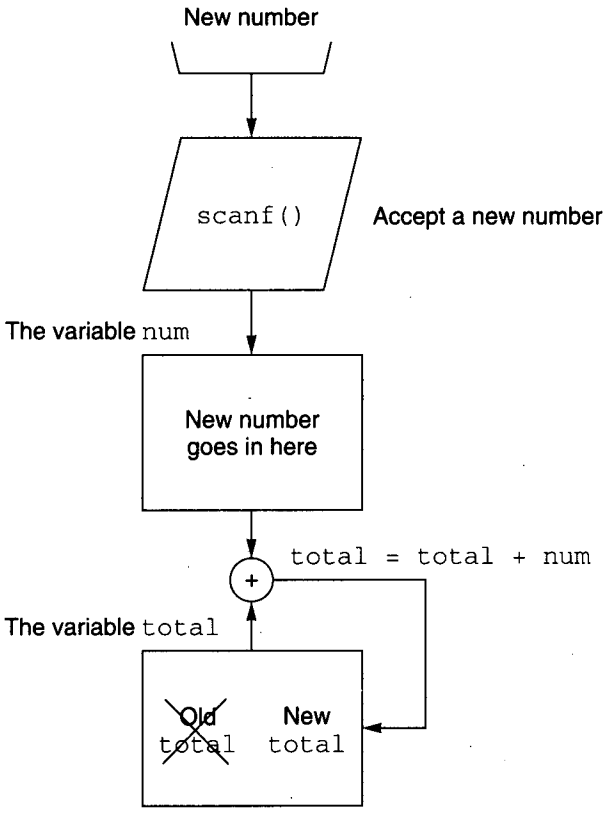


FIGURE 5–3 Accepting and Adding a Number to a Total

all the numbers. After the `while` loop is finished, the last `printf( )` function call is used to display this sum.

Using the same data that was entered in the sample run for Program 5-5, the following sample run of Program 5-6 was made:

```
This program will ask you to enter some numbers.

Enter a number: 26.2
The total is now 26.200000
Enter a number: 5
The total is now 31.200000
Enter a number: 103.456
The total is now 134.656000
Enter a number: 1267.89
The total is now 1402.546000

The final total is 1402.546000
```

Having used an accumulating assignment statement to add the numbers entered, we can now go further and calculate the average of the numbers. Where do we calculate the average—within the `while` loop or outside it?

In the case at hand, calculating an average requires that both a final sum and

Program 5-6

```
#include <stdio.h>
main( )
{
  int count;
  float num, total;

  printf("\nThis program will ask you to enter some numbers.\n");
  count = 1;
  total = 0;

  while (count <= 4)
  {
    printf("\nEnter a number: ");
    scanf("%f", &num);
    total = total + num;
    printf("The total is now %f", total);
    count = count + 1;
  }
  printf("\n\nThe final total is %f",total);
}
```

the number of items in that sum be available. The average is then computed by dividing the final sum by the number of items. At this point, we must ask, "At what point in the program is the correct sum available, and at what point is the number of items available?" In reviewing Program 5-6 we see that the correct sum needed for calculating the average is available after the `while` loop is finished. In fact, the whole purpose of the `while` loop is to ensure that the numbers are entered and added correctly to produce a correct sum. After the loop is finished, we also have a count of the number of items used in the sum. However, due to the way the `while` loop was constructed, the number in `count` (5) when the loop is finished is one more than the number of items (4) used to obtain the total. Knowing this, we simply subtract one from `count` before using it to determine the average. With this as background, see if you can read and understand Program 5-7 (p. 209).

Program 5-7 is almost identical to Program 5-6, except for the calculation of the average. We have also removed the constant display of the total within and after the `while` loop. The loop in Program 5-7 is used to enter and add four numbers. Immediately after the loop is exited, the average is computed and displayed.

Following is a sample run using Program 5-7:

```
This program will ask you to enter some numbers.

Enter a number: 26.2
Enter a number: 5
```

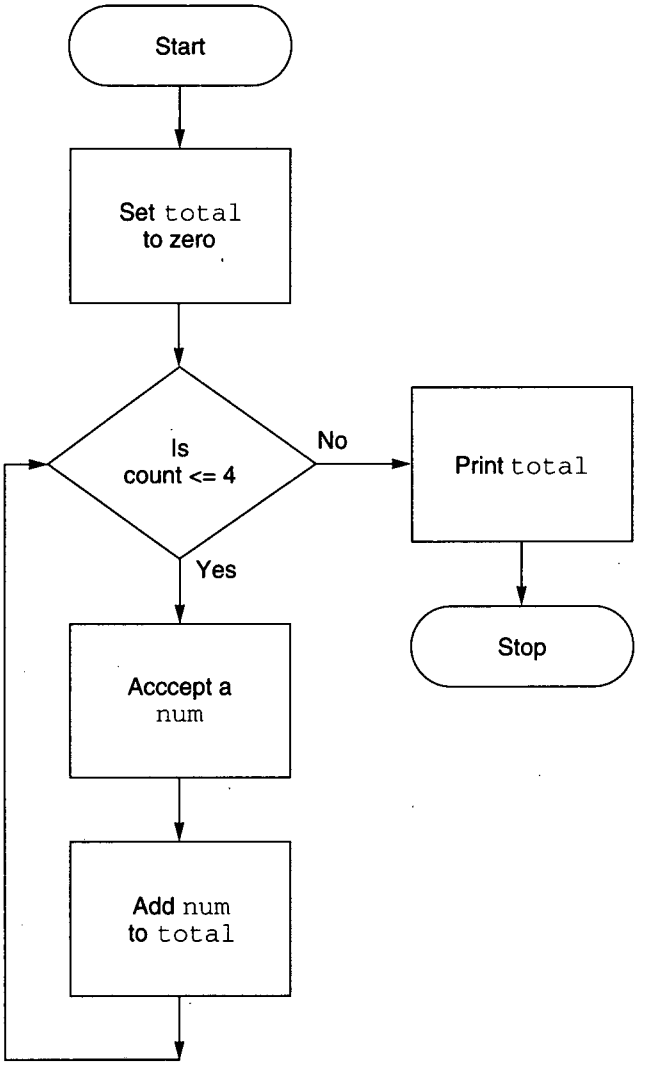


FIGURE 5–4 Accumulation Flow of Control

```
Enter a number: 103.456
Enter a number: 1267.89

The average of the numbers is 350.636500
```

Sentinels

In many situations we do not know the exact number of items to be entered in advance or the items are too numerous to count beforehand. For example, when entering a large amount of market research data we might not want to take the time to count the number of actual data items that are to be entered. In cases like this we want to be able to enter data continuously and, at the end, type in a special data value to signal the end of data input.

In computer programming, data values used to signal either the start or end of a data series are called *sentinels*. The sentinel values, of course, must not con-

Program 5-7

```
#include <stdio.h>
main( )
{
  int count;
  float num, total, average;

  printf("\nThis program will ask you to enter some numbers.\n");
  count = 1;
  total = 0;

  while (count <= 4)
  {
    printf("Enter a number: ");
    scanf("%f", &num);
    total = total + num;
    count = count + 1;
  }
  count = count - 1;
  average = total / count;
  printf("\nThe average of the numbers is %f",average);
}
```

flict with legitimate data values. For example, if we were constructing a program that accepts a student's grades, and assuming that no extra credit is given that could produce a grade higher than 100, we could use any grade higher than 100 as a sentinel value. Program 5-8 (p. 210) illustrates this concept. In Program 5-8 data is continuously requested and accepted until a number larger than 100 is entered. Entry of a number higher than 100 alerts the program to exit the `while` loop and display the sum of the numbers entered.

Following is a sample run using Program 5-8. As long as grades less than or equal to 100 are entered, the program continues to request and accept additional data. When a number greater than 100 is entered, the program adds this number to the total and exits the `while` loop. Outside of the loop and within the `printf( )` function call, the value of the sentinel that was added to the total is subtracted and the sum of the legitimate grades that were entered is displayed.

```
To stop entering grades, type in any number
 greater than 100.

Enter a grade: 95
Enter a grade: 100
Enter a grade: 82
Enter a grade: 101

The total of the grades is 277.000000
```

Program 5-8

```
#include <stdio.h>
main( )
{
  float grade, total;

  grade = 0;
  total = 0;
  printf("\nTo stop entering grades, type in any number");
  printf("\n greater than 100.\n");

  while (grade <= 100)
  {
    printf("Enter a grade: ");
    scanf("%f", &grade);
    total = total + grade;
  }

  printf("\nThe total of the grades is %f",total-grade);
}
```

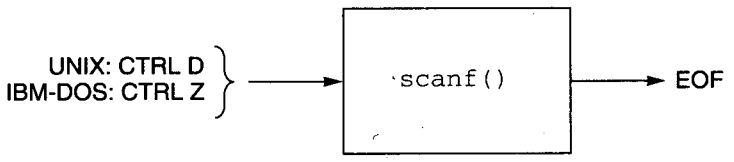
One of the most useful sentinels provided in C is the named constant EOF, which stands for End Of File. The actual value of EOF is compiler dependent, but it is always assigned a code that is not used by any other character. This is how EOF works.

Each computer operating system has its own code for an end-of-file mark. In the UNIX operating system this mark is generated whenever the CTRL and D keys are pressed simultaneously, while in the IBM-DOS operating system the mark is generated whenever the CTRL and Z keys are pressed simultaneously. When a C program detects this combination of keys as an input value, it converts the input value into its own EOF code, as illustrated in Figure 5–5.

The actual definition of the EOF constant, using the `#define` statement previously described in Section 3.4, is available in the `stdio.h` compiler source file included in all C programs. For example, consider Program 5-9.

Notice that the first line in Program 5-9 is the `#include <stdio.h>` statement. Since the `stdio.h` file contains the definition of EOF, this constant may now be referenced in the program.

FIGURE 5–5 Generation of the EOF Constant by the `scanf( )` Function



Program 5-9

```
#include <stdio.h>
main( )

{
  float grade, total = 0;       /* note the initialization here */

  printf("\nTo stop entering grades, press either the F6 key");
  printf("\n or the CTRL and Z keys simultaneously on IBM computers");
  printf("\n or the CTRL and D keys for UNIX operating systems.\n\n");
  printf("Enter a grade: ");

  while ( scanf("%f", &grade) != EOF )
  {
    total = total + grade;
    printf("Enter a grade: ");
  }

  printf("\nThe total of the grades is %f",total);
}
```

EOF is used in Program 5-9 to control the while loop. The expression scanf("%f", &grades) != EOF makes use of the fact that the scanf( ) function returns an EOF value if an attempt is made to read an end-of-file mark. From a user's viewpoint, assuming an IBM computer is being used, pressing both the CTRL and Z keys simultaneously generates an end-of-file mark, which is converted to the EOF constant by scanf( ). Following is a sample run using Program 5-9:

```
To stop entering grades, press either the F6 key
 or the CTRL and Z keys simultaneously on IBM computers
 or the CTRL and D keys for UNIX operating systems.

Enter a grade: 100
Enter a grade: 200
Enter a grade: 300
Enter a grade: ^Z

The total of the grades is 600.000000
```

One distinct advantage of Program 5-9 over Program 5-8 is that the sentinel value is never added into the total, so it does not have to be subtracted later.[1] One disadvantage of Program 5-9, however, is that it requires the user to type in an unfamiliar combination of keys to terminate data input.

[1] This method of input will be very useful when reading data from files rather than from the keyboard. (File input is presented in Chapter 9.)

break and continue Statements

Two useful statements in connection with repetition statements are the `break` and `continue` statements. We have previously encountered the `break` statement in relation to the `switch` statement. The general form of this statement is:

```
break;
```

A `break` statement, as its name implies, forces an immediate break, or exit, from `switch`, `while`, and the `for` and `do-while` statements presented in the next sections.

For example, execution of the following `while` loop is immediately terminated if a number greater than 76 is entered:

```
while(count <= 10)
{
   printf("Enter a number: ");
   scanf("%f", &num);
   if (num > 76)
   {
     printf("You lose!");
     break;        /* break out of the loop */
   }
   else
   printf("Keep on truckin!");
}
/* break jumps to here */
```

The `break` statement violates pure structured programming principles because it provides a second, nonstandard exit from a loop. Nevertheless, the `break` statement is extremely useful and valuable for breaking out of loops when an unusual condition is detected. The `break` statement is also used to exit from a `switch` statement, but this is because the desired `case` has been detected and processed.

The `continue` statement is similar to the `break` statement but applies only to loops created with `while`, `do-while`, and `for` statements. The general format of a `continue` statement is:

```
continue;
```

When `continue` is encountered in a loop, the next iteration of the loop is immediately begun. For `while` loops this means that execution is automatically transferred to the top of the loop and reevaluation of the tested expression is initiated. Although the `continue` statement has no direct effect on a `switch` statement, it can be included within a `switch` statement that itself is contained in a loop. Here the effect of `continue` is the same: the next loop iteration is begun.

As a general rule the `continue` statement is less useful than the `break` statement, but it is convenient for skipping over data that should not be processed while remaining in a loop. For example, invalid grades are simply ignored in the following section of code and only valid grades are added into the total:

```
while (count < 30)
{
  printf("Enter a grade: ");
  scanf("%f", &grade);
  if(grade < 0 || grade > 100) continue;
  total = total + grade;
}
```

The Null Statement

Statements are always terminated by a semicolon. A semicolon with nothing preceding it is also a valid statement, called the null statement. Thus, the statement

```
;
```

is a null statement. This is a do-nothing statement that is used where a statement is syntactically required, but no action is called for. Null statements typically are used with either `while` or `for` statements. An example of a `for` statement using a null statement is found in Program 5-11c in the next section.

Exercises 5.2

1. Rewrite Program 5-6 to compute the total of eight numbers.
2. Rewrite Program 5-6 to display the prompt:

```
Please type in the total number of data values to be added:
```

In response to this prompt, the program should accept a user-entered number and then use this number to control the number of times the `while` loop is executed. Thus, if the user enters `5` in response to the prompt, the program should request the input of five numbers and display the total after five numbers have been entered.

3a. Write a C program to convert Celsius degrees to Fahrenheit. The program should request the starting Celsius value, the number of conversions to be made, and the increment between Celsius values. The display should have appropriate headings and list the Celsius value and the corresponding Fahrenheit value. Use the relationship `Fahrenheit = (9.0 / 5.0) * Celsius + 32.0`.

b. Run the program written in Exercise 3a on a computer. Verify that your program starts at the correct starting Celsius value and contains the exact number of conversions specified in your input data.

4a. Modify the program written in Exercise 3 to request the starting Celsius value, the ending Celsius value, and the increment. Thus, instead of the condition checking for a fixed count, the condition will check for the ending Celsius value.

b. Run the program written in Exercise 4a on a computer. Verify that your output starts at the correct beginning value and ends at the correct ending value.

5. Rewrite Program 5-7 to compute the average of ten numbers.
6. Rewrite Program 5-7 to display this prompt:

```
Please type in the total number of data values to be averaged:
```

In response to this prompt, the program should accept a user-entered number and then use this number to control the number of times the `while` loop is executed. Thus, if the user enters `6` in response to the prompt, the program should request the input of six numbers and display the average of the next six numbers entered.

7. By mistake, a programmer put the statement `average = total / count;` within the `while` loop immediately after the statement `total = total + num;` in Program 5-7. Thus, the `while` loop becomes:

```
while (count <= 4)
{
  printf("Enter a number: ");
  scanf("%f", &num);
  total = total + num;
  average = total / count;
  count = count + 1;
}
```

Will the program yield the correct result with this `while` loop? From a programming perspective, which `while` loop is better to use, and why?

8. In addition to the arithmetic average of a set of numbers both a geometric and harmonic mean can be calculated. The geometric mean of a set of n numbers $x_1, x_2, \ldots x_n$ is defined as:

$$\sqrt[n]{x_1 \cdot x_2 \cdot \cdots \cdot x_n}$$

and the harmonic mean as:

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}}$$

Using these formulas, write a C program that continues to accept numbers until the number 999 is entered, and then calculates and displays both the geometric and harmonic means of the entered numbers. (*Hint:* It will be necessary for your program to correctly count the number of values entered.)

9a. The following data were collected on a recent automobile trip:

	Mileage	Gallons
Start of trip:	22495	Full tank
	22841	12.2
	23185	11.3
	23400	10.5
	23772	11.0
	24055	12.2
	24434	14.7
	24804	14.3
	25276	15.2

Write a C program that accepts a mileage and gallons value and calculates the miles-per-gallon (mpg) achieved for that segment of the trip. The miles-per-gallon is obtained

as the difference in mileage between fillups divided by the number of gallons of gasoline used in the fillup.

b. Modify the program written for Exercise 9a to additionally compute and display the cumulative mpg achieved after each fillup. The cumulative mpg is calculated as the difference between each fillup mileage and the mileage at the start of the trip divided by the sum of the gallons used to that point in the trip.

10a. A bookstore summarizes its monthly transactions by keeping the following information for each book in stock:

Book identification number
Inventory balance at the beginning of the month
Number of copies received during the month
Number of copies sold during the month

Write a C program that accepts this data for each book and then displays the book identification number and an updated book inventory balance using the relationship:

New Balance = Inventory balance at the beginning of the month
+ Number of copies received during the month
− Number of copies sold during the month

Your program should use a `while` loop with a fixed `count` condition so that information on only three books is requested.

b. Run the program written in Exercise 10a on a computer. Review the display produced by your program and verify that the output produced is correct.

11. Modify the program you wrote for Exercise 10a to keep requesting and displaying results until a sentinel identification value of 999 is entered. Run the program on a computer.

12a. The outstanding balance on Rhona Karp's car loan is $8,000. Each month Rhona is required to make a payment of $300, which includes both interest and principal repayment of the car loan. The monthly interest is calculated as .10/12 of the outstanding balance of the loan. After the interest is deducted the remaining part of the payment is used to pay off the loan. Using this information, write a C program that produces a table indicating the beginning monthly balance, the interest payment, the principal payment, and the remaining loan balance after each payment is made. Your output should resemble and complete the entries in the following table until the outstanding loan balance is zero.

Beginning Balance	Interest Payment	Principal Payment	Ending Loan Balance
8000.000000	66.666667	233.333333	7766.666667
7766.666667	64.722223	235.277777	7531.388889
7531.388889	.	.	.
.	.	.	.
.	.	.	.
.	.	.	0.000000

b. Modify the program written in Exercise 12a to display the total of the interest and principal paid at the end of the table produced by your program.

5.3 The for Statement

The `for` statement performs the same functions as the `while` statement, but uses a different form. In many situations, especially those that use a fixed count condition, the `for` statement format is easier to use than its `while` statement equivalent.

The general form of the `for` statement is:

```
for (initializing list; expression; altering list)
  statement;
```

Although the `for` statement looks a little complicated, it is really quite simple if we consider each of its parts separately.

Within the parentheses of the `for` statement are three items, separated by semicolons. Each of these items is optional and can be described individually, but the semicolons must be present.

In its most common form, the initializing list consists of a single statement used to set the starting (initial value) of a counter; the expression contains the maximum or minimum value the counter can have and determines when the loop is finished; and the altering list provides the increment value that is added to or subtracted from the counter each time the loop is executed. Examples of simple `for` statements having this form are:

```
for (count = 1; count < 10; count = count + 1)
  printf("%d ", count);
```

and

```
for (i = 5; i <= 15; i = i + 2)
  printf("%d ", i);
```

In the first `for` statement, the counter variable is named `count`, the initial value assigned to `count` is `1`, the loop continues as long as the value in `count` is less than 10, and the value of `count` is incremented by one each time through the loop. In the next for statement, the counter variable is named `i`, the initial value assigned to `i` is `5`, the loop continues as long as `i`'s value is less than or equal to 15, and the value of `i` is incremented by 2 each time through the loop. In both cases a `printf( )` function call is used to display the value of the counter.

Another example of a `for` loop is contained within Program 5-10.

When Program 5-10 is executed, the following display is produced:

```
NUMBER    SQUARE ROOT
------    -----------
  1        1.000000
  2        1.414214
  3        1.732051
  4        2.000000
  5        2.236068
```

Program 5-10

```
#include <stdio.h>
#include <math.h>
main( )
{
  int count;

  printf("\nNUMBER    SQUARE ROOT");
  printf("\n------    -----------");

  for (count = 1; count <= 5; count = count +1)
    printf("\n  %d          %f",count, sqrt(count));
}
```

The first two lines displayed by the program are produced by the two `printf` statements placed before the `for` statement. The remaining output is produced by the `for` loop. This loop begins with the `for` statement and is executed as follows.

The initial value assigned to the counter variable `count` is `1`. Since the value in `count` does not exceed the final value of `5`, the execution of the `printf` statement within the loop produces the display

```
1        1.000000
```

Control is then transferred back to the `for` statement, which then increments the value in `count` to `2`, and the loop is repeated, producing the display

```
2        1.414214
```

This process continues until the value in `count` exceeds the final value of `5`, producing the complete output table. For comparison purposes, an equivalent `while` loop to the `for` loop contained in Program 5-10 is:

```
count = 1
while (count <= 5)
{
  printf("\n  %d          %f",count, sqrt(count));
  count = count + 1;
 }
```

As seen in this example, the difference between the `for` and `while` loops is the placement of the initialization, condition test, and incrementing items. The grouping together of these items in the `for` statement is very convenient when fixed-count loops must be constructed. See if you can determine the output produced by Program 5-11 (p. 218).

Program 5-11

```
#include <stdio.h>
main( )
{
  int count;

  for (count = 2; count <= 20; count = count + 2)
    printf(%d  ,count);
 }
```

Did you figure it out? The loop starts with a `count` initialized to `2`, stops when `count` exceeds 20, and increments `count` in steps of 2. The output of Program 5-11 is:

```
2  4  6  8  10  12  14  16  18  20
```

The `for` statement does not require that any of the items in parentheses be present or that they be used for initializing or altering the values in the expression statements. However, the two semicolons must be present within the `for`'s parentheses. For example, the construction `for ( ; count <= 20 ;)` is valid.

If the initializing list is missing, the initialization step is omitted when the `for` statement is executed. This, of course, means that the programmer must provide the required initializations before the `for` statement is encountered. Similarly, if the altering list is missing, any expressions needed to alter the evaluation of the tested expression must be included directly within the statement part of the loop. The `for` statement only ensures that all expressions in the initializing list are executed once, before evaluation of the tested expression, and that all expressions in the altering list are executed at the end of the loop before the tested expression is rechecked. Thus, Program 5-11 can be rewritten in any of the three ways shown in Programs 5-11a, 5-11b, and 5-11c.

Program 5-11a

```
#include <stdio.h>
main( )
{
  int count;

  count = 2;     /* initializer outside for statement */
  for ( ; count <= 20; count = count + 2)
    printf("%d ",count);
}
```

Program 5-11b

```
#include <stdio.h>
main( )
{
  int count;

   count = 2;     /* initializer outside for loop */
   for( ; count <= 20; )
   {
     printf("%d ",count);
     count = count + 2;      /* alteration statement */
   }
}
```

Program 5-11c

```
#include <stdio.h>
main( )   /* all expressions within the for's parentheses */
{
  int count;

  for (count = 2; count <= 20; printf("%d  ",count), count = count + 2);
}
```

In Program 5-11a count is initialized outside the for statement and the first list inside the parentheses is left blank. In Program 5-11b, both the initializing list and the altering list are removed from within the parentheses. Program 5-11b also uses a compound statement within the for loop, with the expression-altering statement included in the compound statement. Finally, Program 5-11c has included all items within the parentheses, so there is no need for any useful statement following the parentheses. Here the null statement satisfies the syntactical requirement of one statement to follow the for's parentheses. Observe also in Program 5-11c that the altering list (last set of items in parentheses) consists of two items, and that a comma has been used to separate these items. The use of commas to separate items in both the initializing and altering lists is required if either of these two lists contains more than one item. Last, note the fact that Programs 5-11a, 5-11b, and 5-11c are all inferior to Program 5-11. The for statement in Program 5-11 is much clearer since all the expressions pertaining to the tested expression are grouped together within the parentheses.

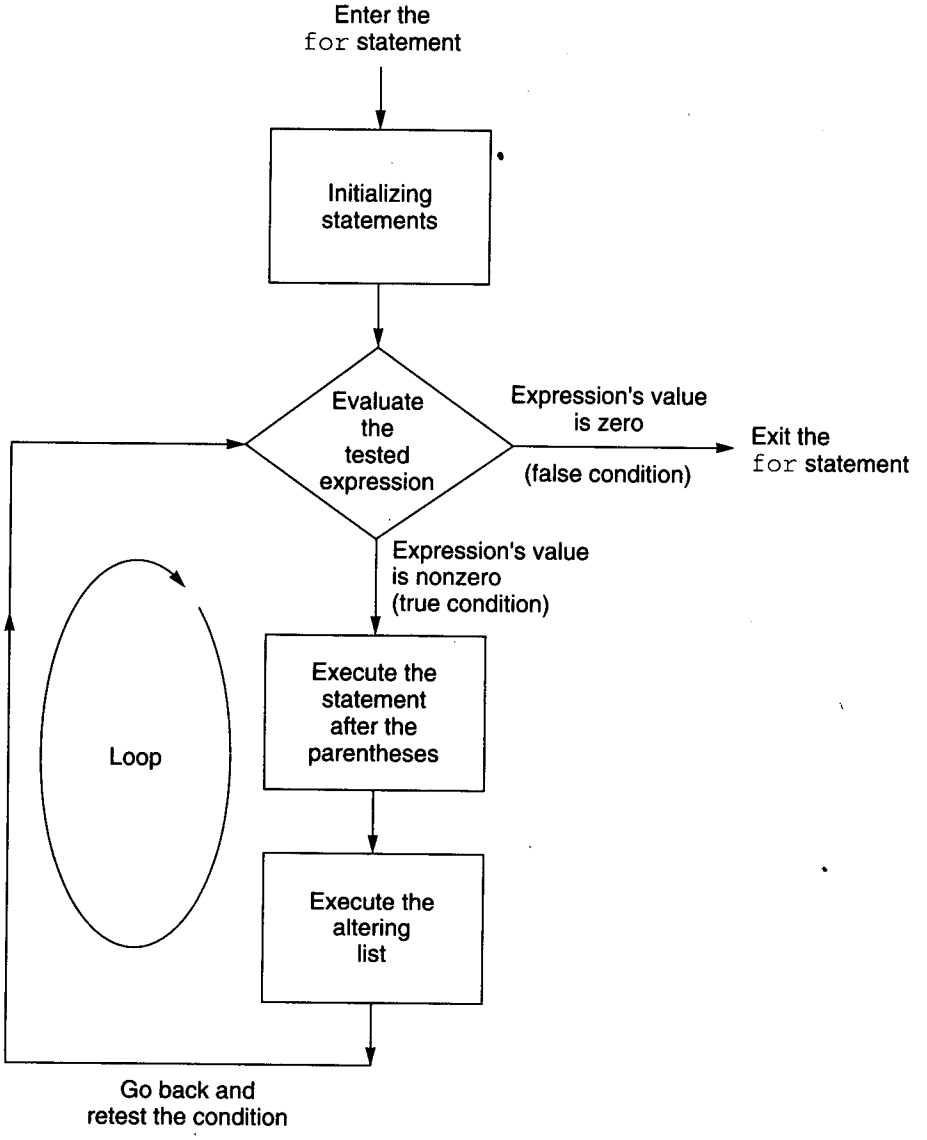
Although the initializing and altering lists can be omitted from a for statement, omitting the tested expression results in an infinite loop. For example, such a loop is created by the statement

```
for (count = 2; ; count = count + 1)
  printf("%d",count);
```

As with the `while` statement, both `break` and `continue` statements can be used within a `for` loop. The `break` forces an immediate exit from the `for` loop, as it does in the `while` loop. The `continue`, however, forces control to be passed to the altering list in a `for` statement, after which the tested expression is reevaluated. This differs from the action of `continue` in a `while` statement, where control is passed directly to the reevaluation of the tested expression.

Figure 5–6 illustrates the internal workings of a `for` loop. As shown, when the `for` loop is completed control is transferred to the first executable statement

FIGURE 5–6 for Loop Flowchart



following the loop. To avoid the necessity of always illustrating these steps, a simplified set of flowchart symbols is available for describing for loops. Using the fact that a for statement can be represented by the flowchart symbol

complete for loops can alternatively be illustrated as shown in Figure 5–7 (p. 222).

To understand the enormous power of for loops, consider the task of printing a table of numbers from 1 to 10, including their squares and cubes, using this statement. Such a table was previously produced using a while loop in Program 5-3. You may wish to review Program 5-3 and compare it to Program 5-12 to get a further sense of the equivalence between for and while loops.

Program 5-12

```
#include <stdio.h>
main( )
{
  int num;

  printf("NUMBER     SQUARE     CUBE\n");
  printf("------     ------     ----\n");

  for (num = 1; num <= 10; ++num)
    printf("%3d         %3d       %4d\n", num, num*num, num*num*num);
}
```

When Program 5-12 is run, the display produced is:

```
NUMBER     SQUARE     CUBE
------     ------     ----
  1          1          1
  2          4          8
  3          9         27
  4         16         64
  5         25        125
  6         36        216
  7         49        343
  8         64        512
  9         81        729
 10        100       1000
```

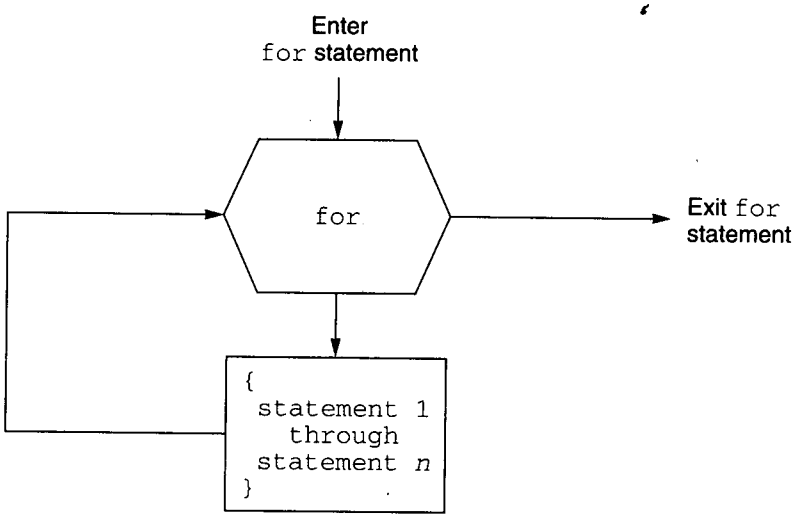


FIGURE 5–7 Simplified `for` Loop Flowchart

Simply changing the number 10 in the `for` statement of Program 5-12 to 1000 creates a loop that is executed 1000 times and produces a table of numbers from 1 to 1000. As with the `while` statement this small change produces an immense increase in the processing and output provided by the program. Notice also that the expression `++num` was used in the altering list in place of the usual `num = num + 1`.

Exercises 5.3

1. Write individual `for` statements for the following cases:
   a. Use a counter named `i` that has an initial value of `1`, a final value of 20, and an increment of 1.
   b. Use a counter named `icount` that has an initial value of `1`, a final value of 20, and an increment of 2.
   c. Use a counter named j that has an initial value of `1`, a final value of 100, and an increment of 5.
   d. Use a counter named `icount` that has an initial value of `20`, a final value of 1, and an increment of −1.
   e. Use a counter named `icount` that has an initial value of `20`, a final value of 1, and an increment of −2.
   f. Use a counter named `count` that has an initial value of `1.0`, a final value of 16.2, and an increment of 0.2.
   g. Use a counter named `xcnt` that has an initial value of `20.0`, a final value of 10.0, and an increment of −0.5.

2. Determine the number of times that each `for` loop is executed for the `for` statements written for Exercise 1.

3. Determine the value in `total` after each of the following loops is executed:
   a.
```
total = 0;
for (i = 1; i <= 10; i = i + 1)
 total = total + 1;
```

b.
```
total = 1;
for (count = 1; count <= 10; count = count + 1)
  total = total * 2;
```
c.
```
total = 0
for ( i = 10; i <= 15; i = i + 1)
  total = total + i;
```
d.
```
total = 50
for (i = 1; i <=10; i = i + 1)
 total = total - i;
```
e.
```
total = 1
for (icnt = 1; icnt <= 8; ++icnt)
 total = total * icnt;
```
f.
```
total = 1.0
for (j = 1; j <= 5; ++j)
 total = total / 2.0;
```

4. Determine the output of the following program:

```
#include <stdio.h>
main( )
{
  int i;

  for (i = 20; i >= 0; i = i - 4)
    printf("%d ",i);
}
```

5. Modify Program 5-12 to produce a table of the numbers zero through 20 in increments of 2, with their squares and cubes.

6. Modify Program 5-12 to produce a table of numbers from 10 to 1, instead of 1 to 10 as it currently does.

7. Write and run a C program that displays a table of 20 temperature conversions from Fahrenheit to Celsius. The table should start with a Fahrenheit value of 20 degrees and be incremented in values of 4 degrees. Recall that Celsius = (5.0/9.0) * (Fahrenheit − 32).

8. Modify the program written for Exercise 7 to initially request the number of conversions to be made.

9. The expansion of a steel bridge as it is heated to a final Celsius temperature, *TF*, from an initial Celsius temperature, *T0*, can be approximated using the formula

$$\text{increase in length} = a * L * (TF\text{-}T0)$$

where a is the coefficient of expansion, which for steel is 11.7 E-6, and L is the length of the bridge at temperature *T0*. Using this formula, write a C program that displays a table of expansion lengths for a steel bridge that is 7365 feet long at zero degrees Celsius, as the temperature increases to 40 degrees in five-degree increments.

10. The probability that an individual telephone call will last less than t minutes can be approximated by the exponential probability function

$$\text{Probability that a call lasts less than } t \text{ minutes} = 1 - e^{-t/a}$$

where a is the average call length and e is Euler's number (2.71828). For example, assuming that the average call length is 2.5 minutes, the probability that a call will last less than one minute is calculated as $1 - e^{-1/2.5} = 0.3297$.

Using this probability function, write a C program that calculates and displays a list of probabilities of a call lasting less than one to less than 10 minutes, in one-minute increments. Assume that the average call length is 5 minutes.

11a. The arrival rate of customers in a busy New York bank can be estimated using the Poisson probability function

$$P(x) = \frac{a^x e^{-a}}{x!}$$

where x = the number of customer arrivals per minute, a = the average number of arrivals per minute, and e = Euler's number (2.71828). For example, if the average number of customers entering the bank is three customers per minute, then a is equal to three. Thus, the probability of 0 customers arriving in any one minute =

$$P(x = 0) + \frac{3^0 e^{-3}}{0!} = .498$$

and the probability of 1 customer arriving in any one minute =

$$P(x = 1) = \frac{3^1 e^{-3}}{1!} = .1494$$

Using the Poisson probability function, write a C program that calculates and displays the probability of 0 to 20 customer arrivals when the average arrival rate is 3 customers per minute.

b. The formula given in Exercise 11a is also applicable for estimating the arrival rate of planes at a busy airport (here, an arriving "customer" is an incoming airplane). Using this same formula modify the program written in Exercise 11a to accept the average arrival rate as an input data item. Then run the modified program to determine the probability of zero to ten planes attempting to land in any one-minute period at an airport during peak arrival times. Assume that the average arrival rate for peak arrival times is two planes per minute.

12. Write and run a program that calculates and displays the amount of money available in a bank account that initially has $1,000 deposited in it and that earns 8 percent interest a year. Your program should display the amount available at the end of each year for a period of ten years. Use the relationship that the money available at the end of each year equals the amount of money in the account at the start of the year plus .08 times the amount available at the start of the year.

13. A machine purchased for $28,000 is depreciated at a rate of $4,000 a year for seven years. Write and run a C program that computes and displays a depreciation table for seven years. The table should have the form:

Depreciation Schedule

Year	Depreciation	End-of-Year Value	Accumulated Depreciation
1	4000	24000	4000
2	4000	20000	8000
3	4000	16000	12000
4	4000	12000	16000
5	4000	8000	20000
6	4000	4000	24000
7	4000	0	28000

14. A manufacturer of widgets has been losing 4 percent of its sales each year. The annual profit for the firm is 10 percent of sales. This year the firm has had $10 million in sales and a profit of $1 million. Determine the expected sales and profit for the next 10 years. Your program should complete and produce a display as follows:

Sales and Profit Projection

Year	Expected Sales	Projected Profit
1	$10000000.00	$1000000.00
2	$ 9600000.00	$ 960000.00
3	.	.
.	.	.
.	.	.
.	.	.
10	_____._____	_____._____
Totals:	$.	$.

5.4 for Loop Programming Techniques

In this section we present four common programming techniques associated with `for` loops.

Technique 1: Interactive Input Within a `for` Loop

Using a `scanf( )` statement inside a `for` loop produces the same effect as when this statement is used inside a `while` loop. For example, in Program 5-13 a `scanf( )` statement is used to input a set of numbers. As each number is input,

Program 5-13

```
#define MAXCOUNT 5
#include <stdio.h>
main( )
/* this program calculates the average of five user-entered numbers */
{
  int count;
  float num, total, average;

  for(total = 0.0, count = 1; count <= MAXCOUNT; ++count)
  {
    printf("Enter a number: ");
    scanf("%f", &num);
    total = total + num;
   }
   average = total / MAXCOUNT;
   printf("\nThe average of the data entered is %f",average);
}
```

it is added to a total. When the `for` loop is exited, the average is calculated and displayed.

The `for` statement in Program 5-13 creates a loop that is executed five times. The user is prompted to enter a number each time through the loop. After each number is entered, it is immediately added to the total. Notice that `total` is initialized to zero when the initializing list of the `for` statement is executed. The loop in Program 5-13 is executed as long as the value in `count` is less than or equal to five, and is terminated when `count` becomes six (the increment to six, in fact, is what causes the loop to end).

Technique 2: Selection Within a `for` Loop

Another common programming technique is to use a `for` loop to cycle through a set of numbers and select those numbers that meet one or more criteria. For example, assume that we want to find both the positive and negative sum of a set of numbers. The criterion here is whether the number is positive or negative, and the logic for implementing this program is given by the pseudocode

```
for each number
{
   Enter a number
   If the number is greater than zero
      add the number to the positive sum
   else
      add the number to the negative sum
}
```

Program 5-14 describes this algorithm in C for the case where five numbers are to be entered.

Following is a sample run using Program 5-14:

```
Enter a number (positive or negative) : 10
Enter a number (positive or negative) : -10
Enter a number (positive or negative) : 5
Enter a number (positive or negative) : -7
Enter a number (positive or negative) : 11

The positive total is 26.000000
The negative total is -17.000000
```

Technique 3: Evaluating Functions of One Variable

A `for` loop can be conveniently constructed to determine and display the values of a single variable mathematical function for a set of values over any specified interval. For example, assume that we want to know the values of the function

$$y = 10x^2 + 3x - 2$$

for x between two and six. Assuming that x has been declared as an integer variable, the following `for` loop can be used to calculate the required values:

Program 5-14

```
#include <stdio.h>
main( )
/* this program computes the positive and negative sums of a set */
/* of five user entered numbers                                  */
{
  int i;
  float usenum, postot, negtot;

  postot = 0; /* this initialization can be done in the declaration */
  negtot = 0; /* this initialization can be done in the declaration */

  for (i = 1; i <= 5; ++i)
  {
    printf("Enter a number (positive or negative) : ");
    scanf("%f", &usenum);
    if (usenum > 0)
      postot = postot + usenum;
    else
      negtot = negtot + usenum;
   }
   printf("\nThe positive total is %f", postot);
   printf("\nThe negative total is %f", negtot);
 }
```

```
for (x = 2; x <= 6; ++x)
{
  y = 10 * pow(x,2.0) + 3 * x - 2;
  printf("\n %3d        %3d", x, y);
}
```

For this loop we have used the variable x as both the counter variable and the unknown (independent variable) in the function. For each value of x from two to six a new value of y is calculated and displayed. This `for` loop is contained within Program 5-15 (p. 228), which also displays appropriate headings for the values printed.

The following is displayed when Program 5-15 is executed:

```
x value   y value
-------   -------
   2         44
   3         97
   4        170
   5        263
   6        376
```

Two items are important here. First, any equation with one unknown can be evaluated using a single `for` loop. The method requires substituting the desired

Program 5-15

```
#include <stdio.h>
#include <math.h>
main( )
{
  int x, y;

  printf("x value   y value");
  printf("\n-------   -------");
  for (x = 2; x <= 6; ++x)
  {
    y = 10 * pow(x,2) + 3 * x - 2;
    printf("\n %3d        %3d", x, y);
  }
}
```

equation into the `for` loop in place of the equation used in Program 5-15, and adjusting the counter values to match the desired solution range.

Second, we are not constrained to using integer values for the counter variable. For example, by specifying a noninteger increment, solutions for fractional values can be obtained. This is shown in Program 5-16, where the equation $y = 10x^2 + 3x - 2$ is evaluated in the range $x = 2$ to $x = 6$ in increments of 0.5.

Program 5-16

```
#include <stdio.h>
#include <math.h>
main( )
{
  float x, y;

  printf("x value      y value");
  printf("\n-------      -------");
  for (x = 2.0; x <= 6.0; x = x + 0.5)
  {
    y = 10.0 * pow(x,2.0) + 3.0 * x - 2.0;
    printf("\n%8.6f    %10.6f", x, y);
  }
}
```

Notice that `x` and `y` have been declared as floating point variables in Program 5-16, to allow these variables to take on fractional values. The following is the output produced by this program:

```
x value     y value
-------     -------
2.000000     44.000000
2.500000     68.000000
3.000000     97.000000
3.500000    131.000000
4.000000    170.000000
4.500000    214.000000
5.000000    263.000000
5.500000    317.000000
6.000000    376.000000
```

Technique 4: Interactive Loop Control

Values used to control a `for` loop may be set using variables rather than constant values. For example, the four statements

```
i = 5;
j = 10;
k = 1;
for (count = i; count <= j; count = count + k)
```

produce the same effect as the single statement

```
for (count = 5; count <= 10; count = count + 1)
```

The advantage of the first `for` statement, where variables are used in the initialization, condition, and altering expression lists, is that it allows us to assign values for these expressions external to the `for` statement. This is especially useful when a `scanf( )` function call is used to set the actual values. To make this a little more tangible, consider Program 5-17.

Program 5-17

```
#include <stdio.h>
main( )
/* this program displays a table of numbers, their squares and cubes */
/* starting from the number 1. The final number in the table is      */
/* input by the user                                                 */
{
  int num, final;

  printf("Enter the final number for the table: ");
  scanf("%d", &final);
  printf("\nNUMBER SQUARE CUBE");
  printf("\n------ ------ ----");
  for (num = 1; num <= final; ++num)
    printf("\n%3d      %3d    %4d", num, num*num, num*num*num);
}
```

In Program 5-17, we have used a variable name within the condition (middle) expression only. Here a `scanf( )` statement has been placed before the `for` statement to allow the user to decide what the final value should be. Notice that this arrangement permits the user to set the size of the table at run time, rather than having the programmer set the table size at compile time. This also makes the program more general, since it now can be used to create a variety of tables without the need for reprogramming and recompiling.

Exercises 5.4

1. **(`scanf( )` Within a Loop)** Write and run a C program that accepts six Fahrenheit temperatures, one at a time, and converts each value entered to its Celsius equivalent before the next value is requested. Use a `for` loop in your program. The conversion required is Celsius = (5.0/9.0) * (Fahrenheit − 32).

2. **(`scanf( )` Within a Loop)** Write and run a C program that accepts ten individual values of gallons, one at a time, and converts each value entered to its liter equivalent before the next value is requested. Use a `for` loop in your program. The conversion required is liters = 3.785 * gallons.

3. **(Interactive Loop Control)** Modify the program written for Exercise 2 to initially request the number of data items that will be entered and converted.

4. **(Interactive Loop Control)** Modify Program 5-14 so that the number of entries to be in `input` is specified by the user when the program is executed.

5. **(Selection)** Modify Program 5-14 so that it displays the average of the positive and negative numbers. (*Hint:* Be careful not to count the number zero as a negative number.) Test your program by entering the numbers 17, −10, 19, 0, −4. The positive average displayed by your program should be 18.5 and the negative average −7.

6a. **(Selection)** Write a C program that selects and displays the maximum value of five numbers that are to be entered when the program is executed. (*Hint:* Use a `for` loop with both a `scanf( )` and an `if` statement internal to the loop.)

b. Modify the program written for Exercise 6a so that it displays both the maximum value and the position in the input set of numbers where the maximum occurs.

7. **(Mathematical Functions)** Modify Program 5-16 to produce a table of y values for the following:

a. $y = 3x^5 - 2x^3 + x$ for x between 5 and 10 in increments of .2

b. $y = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}$ for x between 1 and 3 in increments of .1

c. $y = 2e^{.8t}$ for t between 4 and 10 in increments of .2

8. **(Mathematical Functions)** A model of worldwide population in billions of people is given by the equation

$$\text{Population} = 4.88(1 + e^{.02t})$$

where t is the time in years ($t = 0$ represents January 1985 and $t = 1$ represents January 1986). Using this formula, write a C program that displays a yearly population table for the years 1988 through 1994.

9. **(Mathematical Functions)** The x and y coordinates, as a function of time, t, of a projectile fired with an initial velocity v at an angle of θ with respect to the ground is given by

$$x = v\,t\cos(\theta)$$
$$y = v\,t\sin(\theta)$$

Using these formulas, write a C program that displays a table of x and y values for a projectile fired with an initial velocity of 500 ft/sec at an angle of 22.8 degrees. (*Hint:* Remember to convert to radian measure.) The table should contain values corresponding to the time interval 0 to 10 seconds in increments of one-half seconds.

10. **(Interactive Loop Control)** Modify Program 5-17 to accept the starting and increment values of the table produced by the program.

11. **(Interactive Loop Control)** Write a C program that converts Fahrenheit to Celsius temperature in increments of 5 degrees. The initial value of the Fahrenheit temperature and the total conversions to be made are to be requested as user input during program execution. Recall that Celsius = (5.0/9.0) * (Fahrenheit − 32.0).

5.5 Nested Loops

In many situations it is convenient to have a loop contained within another loop. Such loops are called nested loops. A simple example of a nested loop is:

```
for(i = 1; i <= 5; ++i)          /* start of outer loop <------+ */
{                                /*                         | */
  printf("\ni is now %d\n",i);   /*                         | */
  for(j = 1; j <= 4; ++j)        /* start of inner loop <--+ | */
    printf("  j = %d", j);       /* end of inner loop  <---+ | */
}                                /* end of outer loop  <-------+ */
```

The first loop, controlled by the value of `i`, is called the outer loop. The second loop, controlled by the value of `j`, is called the inner loop. Notice that all statements in the inner loop are contained within the boundaries of the outer loop and that we have used a different variable to control each loop. For each single trip through the outer loop, the inner loop runs through its entire sequence. Thus, each time the `i` counter increases by 1, the inner `for` loop executes completely. This situation is illustrated in Figure 5–8 (p. 232).

Program 5-18 (p. 232) includes the above code in a working program.

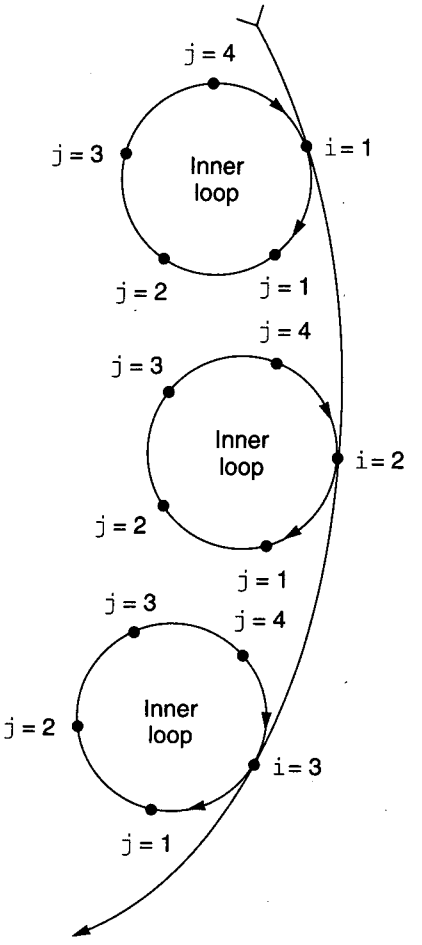
Following is the output of a sample run of Program 5-18:

```
i is now 1
  j = 1  j = 2  j = 3  j = 4
i is now 2
  j = 1  j = 2  j = 3  j = 4
i is now 3
  j = 1  j = 2  j = 3  j = 4
i is now 4
  j = 1  j = 2  j = 3  j = 4
i is now 5
  j = 1  j = 2  j = 3  j = 4
```

Program 5-18

```
#include <stdio.h>
main( )
{
  int i,j;

  for(i = 1; i <= 5; ++i)        /* start of outer loop <------+ */
  {                              /*                            | */
    printf("\ni is now %d\n",i);/*                            | */
    for(j = 1; j <= 4; ++j)      /* start of inner loop <--    | */
      printf("  j = %d", j);     /* end of inner loop  <---    | */
  }                              /* end of outer loop  <-------+ */
}
```

FIGURE 5–8 For Each `i, j` Loops



Let us use a nested loop to compute the average grade for each student in a class of 20 students. Each student has taken four exams during the course of the semester. The final grade is calculated as the average of these examination grades.

Program Analysis

The pseudocode for this example is:

```
for 20 times
{
   set student grade total to zero
   for 4 times
   {
      input a grade
      add the grade to the total
   } /* end of inner for loop */
   calculate student's average grade
   print the student's average grade
} /* end of outer for loop */
```

As described by the pseudocode, an outer loop consisting of 20 passes will be used to compute the average grade for each student. The inner loop will consist of 4 passes. One examination grade is entered in each inner loop pass. As each grade is entered it is added to the total for the student, and at the end of the loop the average is calculated and displayed. Program 5-19 uses a nested loop to make the required calculations.

Program 5-19

```
#include <stdio.h>
main( )
{
  int i,j;
  float grade, total, average;

  for (i = 1; i <= 20; ++i)     /* start of outer loop */
  {
    total = 0;                  /* clear the total for this student */
    for ( j = 1; j <= 4; ++j)   /* start of inner loop */
    {
      printf("Enter an examination grade for this student: ");
      scanf("%f", &grade);
      total = total + grade;    /* add the grade into the total */
    }                           /* end of the inner for loop    */
    average = total / 4;        /* calculate the average        */
    printf("\n\nThe average for student %d is %f",i,average);
  }                             /* end of the outer for loop    */
}
```

In reviewing Program 5-19, pay particular attention to the initialization of `total` within the outer loop, before the inner loop is entered. The variable `total` is initialized 20 times, once for each student. Also notice that the average is calculated and displayed immediately after the inner loop is finished. Since the statements that compute and print the average are also contained within the outer loop, 20 averages are calculated and displayed. The entry and addition of each grade within the inner loop use techniques we have seen before, which should now be familiar to you.

Exercises 5.5

1. Four experiments are performed, each experiment consisting of six test results. The results for each experiment are given below. Write a program using a nested loop to compute and display the average of the test results for each experiment.

1st experiment results:	23.2	31.5	16.9	27.5	25.4	28.6
2nd experiment results:	34.8	45.2	27.9	36.8	33.4	39.4
3rd experiment results:	19.4	16.8	10.2	20.8	18.9	13.4
4th experiment results:	36.9	39.5	49.2	45.1	42.7	50.6

2. Modify the program written for Exercise 1 so that the number of test results for each experiment is entered by the user. Write your program so that a different number of test results can be entered for each experiment.

3. a. A bowling team consists of five players. Each player bowls three games. Write a C program that uses a nested loop to enter each player's individual scores and then computes and displays the average score for each bowler. Assume that each bowler has the following scores:

1st bowler:	286	252	265
2nd bowler:	212	186	215
3rd bowler:	252	232	216
4th bowler:	192	201	235
5th bowler:	186	236	272

b. Modify the program written for Exercise 3a to calculate and display the average team score. (*Hint:* Use a second variable to store the total of all the players' scores.)

4. Rewrite the program written for Exercise 3a to eliminate the inner loop. To do this, you will have to input three scores for each bowler rather than one at a time.

5. Write a program that calculates and displays values for *Y* when

$$Y = X3Z/(X\text{-}Z)$$

Your program should calculate *Y* for values of *X* ranging between 1 and 5 and values of *Z* ranging between 2 and 6. *X* should control the outer loop and be incremented in steps of 0.2 and *Z* should be incremented in steps of 0.5. Your program should also display the message `FUNCTION UNDEFINED` when the *X* and *Z* values are equal.

6. Write a program that calculates and displays the yearly amount available if $1,000 is invested in a bank account for 10 years. Your program should display the amounts available for interest rates from 6 percent to 12 percent inclusively, at 1 percent increments. Use a nested loop, with the outer loop having a fixed count of 7 and the inner

loop a fixed count of 10. The first iteration of the outer loop should use an interest rate of 6 percent and display the amount of money available at the end of the first 10 years. In each subsequent pass through the outer loop, the interest rate should be increased by 1 percent. Use the relationship that the money available at the end of each year equals the amount of money in the account at the start of the year, plus the interest rate times the amount available at the start of the year.

5.6 The do Statement

Both the `while` and `for` statements evaluate an expression at the start of the repetition loop. In some cases, however, it is more convenient to test the expression at the end of the loop. For example, suppose we have constructed the following `while` loop to calculate sales taxes:

```
printf("Enter a price: ");
scanf("%f", &price);
while (price != SENTINEL)
{
  salestax = RATE * price;
  printf("The sales tax is $%5.2f",salestax);
  printf("\nEnter a price: ");
  scanf("%f", &price);
}
```

Using this `while` statement requires either duplicating the prompt and `scanf( )` function calls before the loop and then within the loop, as we have done, or resorting to some other artifice to force initial execution of the statements within the `while` loop.

The `do` statement, as its name implies, allows us to do some statements before an expression is evaluated. In many situations this can be used to eliminate the duplication illustrated in the previous example. The general form of the `do` statement is:

```
do
  statement;
while (expression);
```
← don't forget the final ;

As with all C programs, the single statement in the `do` may be replaced with a compound statement. A flow-control diagram illustrating the operation of the `do` statement is shown in Figure 5–9.

As illustrated in Figure 5–9, all statements within the `do` statement are executed at least once before the expression is evaluated. Then, if the expression has a nonzero value, the statements are executed again. This process continues until the expression evaluates to zero. For example, consider the following `do` statement:

```
do
{
  printf("\nEnter a price: ");
  scanf("%f", &price);
  salestax = RATE * price;
  printf("The sales tax is $%5.2f", salestax);
}
while (price != SENTINEL);
```

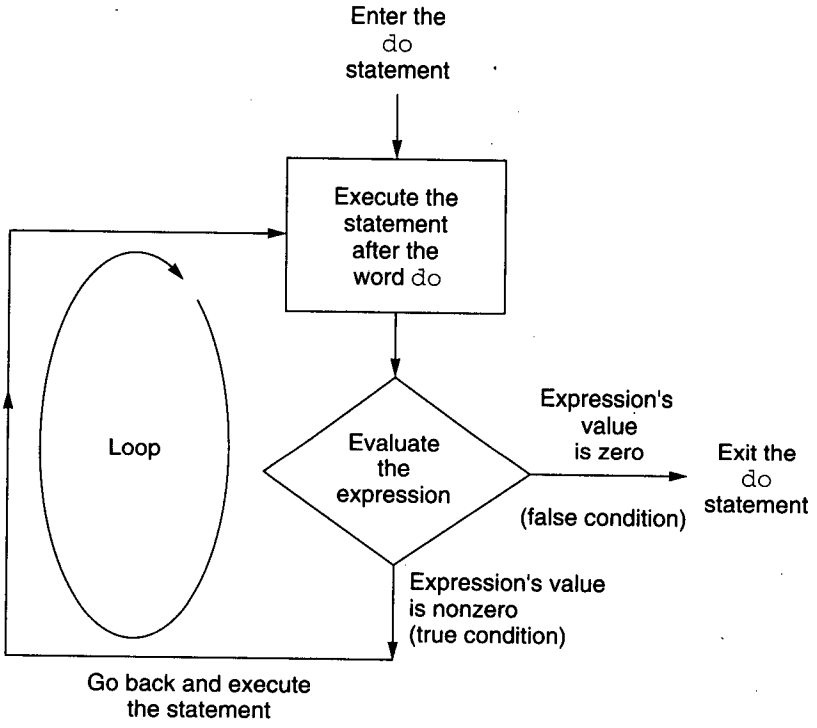
Observe that only one prompt and `scanf( )` statement are required because the tested expression is evaluated at the end of the loop.

As with all repetition statements, the `do` statement can always replace or be replaced by an equivalent `while` or `for` statement. The choice of which statement to use depends on the application and the style preferred by the programmer. In general, the `while` and `for` statements are preferred because they clearly let anyone reading the program know what is being tested "right up front" at the top of the program loop.

Validity Checks

The `do` statement is particularly useful in filtering user-entered input and providing data validity checks. For example, assume that an operator is required to enter a valid customer identification number between the numbers 1000 and 1999. A number outside this range is to be rejected and a new request for a valid

FIGURE 5–9 The `do` Statement's Flow of Control



number made. The following section of code provides the necessary data filter to verify the entry of a valid identification number:

```
do
{
  printf("\nEnter an identification number: ");
  scanf("%f", &id_num);
}
while (id_num < 1000 || id_num > 1999);
```

Here, a request for an identification number is repeated until a valid number is entered. This section of code is "bare bones" in that it neither alerts the operator to the cause of the new request for data nor allows premature exit from the loop if a valid identification number cannot be found. An alternative removing the first drawback is:

```
do
{
  printf("\nEnter an identification number: ");
  scanf("%f", &id_num);
  if (id_num < 1000 || id_num > 1999)
  {
    printf("\n An invalid number was just entered");
    printf("\nPlease check the ID number and re-enter");
  }
  else
    break;    /* break if a valid id num was entered */
} while(1);   /* this expression is always true */
```

Here we have used a `break` statement to exit from the loop. Since the expression being evaluated by the `do` statement is always 1 (true), an infinite loop has been created that is only exited when the `break` statement is encountered.

Exercises 5.6

1 *a.* Using a `do` statement, write a C program to accept a grade. The program should request a grade continuously as long as an invalid grade is entered. An invalid grade is any grade less than 0 or greater than 100. After a valid grade has been entered, your program should display the value of the grade entered.
b. Modify the program written for Exercise 1a so that the user is alerted when an invalid grade has been entered.
c. Modify the program written for Exercise 1b so that it allows the user to exit the program by entering the number 999.
d. Modify the program written for Exercise 1b so that it automatically terminates after five invalid grades are entered.

2 *a.* Write a C program that continuously requests a grade to be entered. If the grade is less than 0 or greater than 100, your program should print an appropriate message informing the user that an invalid grade has been entered, else the grade should be

added to a total. When a grade of 999 is entered the program should exit the repetition loop and compute and display the average of the valid grades entered.
b. Run the program written in Exercise 2a on a computer and verify the program using appropriate test data.

3 *a.* Write a C program to reverse the digits of a positive integer number. For example, if the number 8735 is entered, the number displayed should be 5378. (*Hint:* Use a `do` statement and continuously strip off and display the units digit of the number. If the variable `num` initially contains the number entered, the units digit is obtained as `(num % 10)`. After a units digit is displayed, dividing the number by 10 sets up the number for the next iteration. Thus, `(8735 % 10)` is `5` and `(8735 / 10)` is `873`. The `do` statement should continue as long as the remaining number is not zero.)
b. Run the program written in Exercise 3a on a computer and verify the program using appropriate test data.

4. Repeat any of the exercises in Section 5.3 using a `do` statement rather than a `for` statement.

5.7 Common Programming Errors

Five errors are commonly made by beginning C programmers when using repetition statements. Two of these pertain to the tested expression and have already been encountered with the `if` and `switch` statements. The first is the inadvertent use of the assignment operator, `=`, for the equality operator, `==`, in the tested expression. An example of this error is typing the assignment expression `a = 5` instead of the desired relational expression `a == 5`. Since the tested expression can be any valid C expression, including arithmetic and assignment expressions, this error is not detected by the compiler.

As with the `if` statement, repetition statements should not use the equality operator, `==`, when testing floating point or double precision operands. For example, the expression `fnum == .01` should be replaced by an equivalent test requiring that the absolute value of `fnum - .01` be less than an acceptable amount. The reason for this is that all numbers are stored in binary form. Using a finite number of bits, decimal numbers such as .01 have no exact binary equivalent, so that tests requiring equality with such numbers can fail.

The next two errors are particular to the `for` statement. The most common is to place a semicolon at the end of the `for`'s parentheses, which frequently produces a do-nothing loop. For example, consider the statements

```
for(count = 0; count < 10; ++ count);
  total = total + num;
```

Here the semicolon at the end of the first line of code is a null statement. This has the effect of creating a loop that is executed 10 times with nothing done except the incrementing and testing of count. This error tends to occur because C programmers are used to ending most lines with a semicolon.

The next error occurs when commas are used to separate the items in a `for` statement instead of the required semicolons. An example of this is the statement

```
for (count = 1, count < 10, ++count)
```

Commas must be used to separate items within the initializing and altering lists, and semicolons must be used to separate these lists from the tested expression.

The last error occurs when the final semicolon is omitted from the `do` statement. This error is usually made by programmers who have learned to omit the semicolon after the parentheses of a `while` statement and carry over this habit when the reserved word `while` is encountered at the end of a `do` statement.

5.8 Chapter Summary

1. The `while`, `for`, and `do` repetition statements create program loops. These statements evaluate an expression and, based on the resulting expression value, either terminate the loop or continue with it.
2. The `while` statement checks its expression before any other statement in the loop. This requires that any variables in the tested expression have values assigned before `while` is encountered. Within a `while` loop there must be a statement that alters the tested expression's value.
3. The `for` statement is extremely useful in creating loops that must be executed a fixed number of times. Initializing expressions, the tested expression, and expressions affecting the tested expression can all be included in parentheses at the top of a `for` loop. Additionally, any other loop statement can be included within the `for`'s parentheses as part of its altering list.
4. The `do` statement checks its expression at the end of the loop. This ensures that the body of a `do` loop is executed at least once. Within a `do` loop there must be at least one statement that alters the tested expression's value.

Arrays

Chapter Six

6.1 One-Dimensional Arrays
6.2 Array Initialization
6.3 Two-Dimensional Arrays
6.4 Applications
6.5 Common Programming Errors
6.6 Chapter Summary
6.7 Enrichment Study: Sorting Methods

The variables used so far have all had a common characteristic: Each variable could only be used to store a single value at a time. For example, although the variables `key`, `count`, and `grade` declared in the statements

```
char key;
int count;
float grade;
```

are of different data types, each variable can only store one value of the declared data type. These types of variables are called scalar variables. A *scalar variable* is a single variable that cannot be further subdivided or separated into a legitimate data type.

Frequently we may have a set of values, all of the same data type, that form a logical group. For example, Figure 6–1 illustrates three groups of items. The first group is a list of five floating point temperatures, the second group is a list of four character codes, and the last group is a list of six integer voltages.

A simple list containing individual items of the same scalar data type is called a *one-dimensional array*. In this chapter we describe how one-dimensional arrays are declared, initialized, stored inside a computer, and used. Additionally, we explore the use of one-dimensional arrays with example programs and present the procedures for declaring and using multi-dimensional arrays.

6.1 One-Dimensional Arrays

A one-dimensional array, which is also referred to as either a single-dimensional array, a list, or a vector, is a group of related values with the same data type that is stored using a single group name. In C, as in other computer languages, the group name is referred to as the array name. For example, consider the list of temperatures illustrated in Figure 6–2.

All the temperatures in the list are floating point numbers and must be declared as such. However, the individual items in the list do not have to be declared separately. The items in the list can be declared as a single unit and stored under a common variable name called the array name. For convenience, we will choose `temp` as the name for the list shown in Figure 6–2. To specify that `temp` is to store five individual floating point values requires the declaration statement `float temp[5]`. Notice that this declaration statement gives the data

FIGURE 6–1 Three Lists of Items

Temperatures	Codes	Voltages
95.75	Z	12
83.0	C	5
97.625	K	3
72.5	L	55
86.25		16
		6

Temperatures
95.75
83.0
97.625
72.5
86.25

FIGURE 6–2 A List of Temperatures

type of the items in the array, the array (or list) name, and the number of items in the array. Further examples of array declarations are:

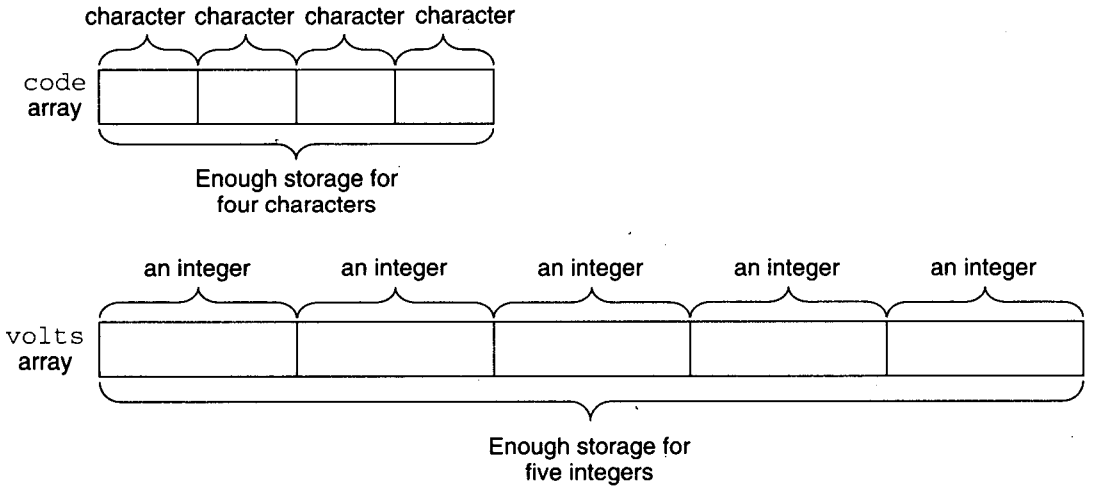
```
int volts[5];
char code[4];
float amount[100];
```

Each array has sufficient memory reserved for it to hold the number of data items given in the declaration statement. Thus, the array named `volts` has storage reserved for five integers, the array named `code` has storage reserved for four characters, and the array named `amount` has storage reserved for 100 floating point numbers. Figure 6–3 illustrates the storage reserved for the `code` and `volts` arrays.

Each item in an array is called an *element* or *component* of the array. The individual elements stored in the arrays illustrated in Figure 6–3 are stored sequentially, with the first array element stored in the first reserved location, the second element stored in the second reserved location, and so on until the last element is stored in the last reserved location.

To access individual elements in an array requires some unique means of identifying each element. Since elements in the array are stored sequentially, any individual element can be accessed by giving the name of the array and the element's position. This position is called the element's *subscript* or *index* value (the

FIGURE 6–3 The code and volts Arrays in Memory



two terms are synonymous). For a one-dimensioned array the first element has a subscript of 0, the second element has a subscript of 1, and so on. In C, the array name and subscript of the desired element are combined by listing the subscript in braces after the array name. For example, given the declaration `float temp[5]`,

`temp[0]` refers to the first temperature stored in the `temp` array
`temp[1]` refers to the second temperature stored in the `temp` array
`temp[2]` refers to the third temperature stored in the `temp` array
`temp[3]` refers to the fourth temperature stored in the `temp` array
`temp[4]` refers to the fifth temperature stored in the `temp` array

Figure 6–4 illustrates the `temp` array in memory with the correct designation for each array element. Each individual element is called an *indexed variable* or a *subscripted variable* (the two terms are synonomous), since both a variable name and a subscript or index value must be used to reference the element. Remember that the index or subscript value gives the position of the element in the array.

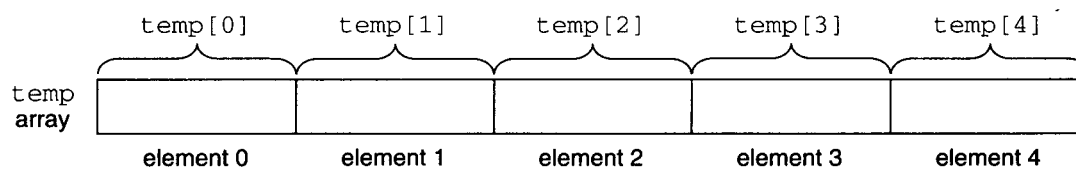
The subscripted variable, `temp[0]`, is read as "temp sub zero." This is a shortened way of saying "the temp array subscripted by zero," and distinguishes the first element in an array from a scalar variable that could be declared as `temp0`. Similarly, `temp[1]` is read as "temp sub one," `temp[2]` as "temp sub two," `temp[3]` as "temp sub three," and `temp[4]` as "temp sub four."

Although it may seem unusual to reference the first element with an index of zero, doing so increases the computer's speed of accessing array elements. Internally, unseen by the programmer, the computer uses the index as an offset from the array's starting position. As illustrated in Figure 6–5, the index tells the computer how many elements to skip over, starting from the beginning of the array, to get to the desired element. From a programmer's viewpoint, then, it is preferable to think of the first element in the array as element zero, the second element as element one, and so on. This makes the index value agree with the element number.

Subscripted variables can be used anywhere scalar variables are valid. Examples using the elements of the `temp` array are:

```
temp[0] = 95.75;
temp[1] = temp[0] - 11.0;
temp[2] = 5.0 * temp[0];
temp[3] = 79.0;
temp[4] = (temp[1] + temp[2] - 3.1) / 2.2;
sum = temp[0] + temp[1] + temp[2] + temp[3] + temp[4];
```

FIGURE 6–4 Identifying Individual Array Elements



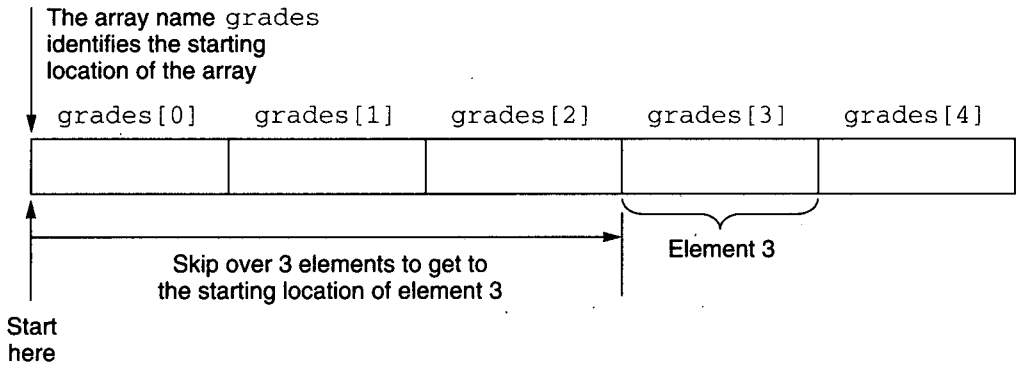


FIGURE 6–5 Accessing an Individual Array Element—Element 3

The subscript contained within braces need not be an integer constant; any expression that evaluates to an integer may be used as a subscript.[1] In each case, of course, the value of the expression must be within the valid subscript range defined when the array is declared. For example, assuming that `i` and `j` are `int` variables, the following subscripted variables are valid:

```
temp[i]
temp[2*i]
temp[j-i]
```

One extremely important advantage of using integer expressions as subscripts is that it allows sequencing through an array by using a loop. This makes statements like the following unnecessary:

```
sum = temp[0] + temp[1] + temp[2] + temp[3] + temp[4];
```

The subscript values in this statement can be replaced by a `for` loop counter to access each element in the array sequentially. For example, the code

```
sum = 0;                        /* initialize the sum to zero */
for (i = 0 i <= 4 ++i)
  sum = sum + temp[i];          /* add in a temperature */
```

sequentially retrieves each array element and adds the element to `sum`. Here the variable `i` is used both as the counter in the `for` loop and as a subscript. As `i` increases by one each time through the loop, the next element in the array is referenced. The procedure for adding the array elements within the `for` loop is similar to the accumulation procedure we have used many times before.

The advantage of using a `for` loop to sequence through an array becomes apparent when working with larger arrays. For example, if the `temp` array contained 100 values rather than just 5, simply changing the number 4 to 99 in the

[1] Some compilers permit floating point variables as subscripts; in these cases the floating point value is truncated to an integer value.

for statement is sufficient to sequence through the 100 elements and add each temperature to the sum.

As another example of using a for loop to sequence through an array, assume that we want to locate the maximum value in an array of 1000 elements named volts. The procedure we will use to locate the maximum value is to assume initially that the first element in the array is the largest number. Then, as we sequence through the array, the maximum is compared to each element. When an element with a higher value is located, that element becomes the new maximum. The following code does the job:

```
maximum = volts[0];          /* set the maximum to element zero      */
for (i = 1; i <= 999; ++i) /* cycle through the rest of the array */
  if (volts[i] > maximum)  /* compare each element to the maximum */
    maximum = volts[i];    /* capture the new high value           */
```

In this code the for statement consists of one if statement. The search for a new maximum value starts with the element 1 of the array and continues through the last element. In a 1000-element array, the last element is 999. Each element is compared to the current maximum, and when a higher value is encountered it becomes the new maximum.

Input and Output of Array Values

Individual array elements can be assigned values interactively using the scanf( ) function. Examples of individual data entry statements are:

```
scanf("%f", &temp[0]);
scanf("%f %f %f", &temp[1], &temp[2], &temp[3]);
scanf("%f %f", &temp[4], &volts[6]);
```

In the first statement a single value will be read and stored in the variable named temp[0]. The second statement causes three values to be read and stored in the variables temp[1], temp[2], and temp[3], respectively. Finally, the last scanf( ) statement can be used to read values into the variables temp[4] and volts[6].

Alternatively, a for loop can be used to cycle through the array for interactive data input. For example, the code

```
for (i = 0; i < = 4; ++i)
{
  printf("Enter a temperature: ");
  scanf("%f", &temp[i]);
}
```

prompts the user for five temperatures. The first temperature entered is stored in temp[0], the second temperature entered in temp[1], and so on until five temperatures have been input.

One caution should be mentioned about storing data in an array. C does not check the value of the index being used (called a *bounds check*). If an array has been declared as consisting of 10 elements, for example, and you use an index of 12, which is outside the bounds of the array, C will not notify you of the error

when the program is compiled. The program will attempt to access element 12 by skipping over the appropriate number of bytes from the start of the array. Usually this results in a program crash—but not always. If the referenced location itself contains a data value, the program will simply access the value in the referenced memory locations. This leads to more errors, which are particularly troublesome to locate when the variable legitimately assigned to the storage location is used at a different point in the program.

During output, individual array elements can be displayed using the `printf( )` function or complete sections of the array can be displayed by including a `printf( )` function call within a `for` loop. Examples using `printf( )` to display subscripted variables are:

```
printf("%f", volts[6]);
printf("The value of element %d is %f", i, temp[i]);
for (n = 5; n <= 20; ++n)
  printf("\n%d %f", n, amount[n]);
```

The first `printf( )` statement displays the value of the subscripted variable `volts[6]`. The second `printf( )` statement displays the value of the subscript `i` and the value of `temp[i]`. Before this statement can be executed, `i` would have to have an assigned value. Finally, the last example includes a `printf( )` statement within a `for` loop. Both the value of the index and the value of the elements from 5 to 20 are displayed.

Program 6-1 illustrates these input and output techniques using an array named `grades` that is defined to store five integer numbers. Included in the program are two `for` loops. The first `for` loop is used to cycle through each array element and allows the user to input individual array values. After five values have been entered, the second `for` loop is used to display the stored values.

Program 6-1

```
#include <stdio.h>
main( )
{
  int i, grades[5];

  for (i = 0; i <= 4; ++i)          /* Enter five grades */
  {
    printf("Enter a grade: ");
    scanf("%d", &grades[i]);
  }

  for (i = 0; i <= 4; ++i)          /* Print five grades */
    printf("\ngrades[%d] is %d", i, grades[i]);
}
```

Following is a sample run using Program 6-1:

```
Enter a grade: 85
Enter a grade: 90
Enter a grade: 78
Enter a grade: 75
Enter a grade: 92

grades[0] is 85
grades[1] is 90
grades[2] is 78
grades[3] is 75
grades[4] is 92
```

In reviewing the output produced by Program 6-1, pay particular attention to the difference between the subscript value displayed and the numerical value stored in the corresponding array element. The subscript value refers to the location of the element in the array, while the subscripted variable refers to the value stored in the designated location.

In addition to simply displaying the values stored in each array element, the elements can also be processed by appropriately referencing the desired element. For example, in Program 6-2 the value of each element is accumulated in a total, which is displayed upon completion of the individual display of each array element.

Program 6-2

```
#include <stdio.h>
main( )
{
  int i, grades[5], total = 0;

  for (i = 0; i <= 4; ++i)        /* Enter five grades */
  {
    printf("Enter a grade: ");
    scanf("%d", &grades[i]);
  }

  printf("\nThe total of the grades ");

  for (i = 0; i <= 4; ++i) /* Display and total the grades */
  {
     printf("%d   ", grades[i]);
     total =  total + grades[i];
  }
  printf("is  %d",total);
}
```

Following is a sample run using Program 6-2:

```
Enter a grade: 85
Enter a grade: 90
Enter a grade: 78
Enter a grade: 75
Enter a grade: 92

The total of the grades 85  90  78  75  92  is  420
```

Notice that in Program 6-2, unlike Program 6-1, only the values stored in each array element are displayed. Although the second `for` loop was used to accumulate the total of each element, the accumulation could also have been accomplished in the first loop by placing the statement `total = total + grades[i];` after the `scanf( )` call used to enter a value. Also notice that the `printf( )` call used to display the total is made outside the second `for` loop, so that the total is displayed only once, after all values have been added to the total. If this `printf( )` call were placed inside the `for` loop five totals would be displayed, with only the last displayed total containing the sum of all the array values.

Exercises 6.1

1. Write array declarations for the following:
- **a.** a list of 100 floating point voltages
- ***b.*** a list of 50 floating point temperatures
- ***c.*** a list of 30 characters, each representing a code
- ***d.*** a list of 100 integer years
- ***e.*** a list of 32 floating point velocities
- ***f.*** a list of 1000 floating point distances
- ***g.*** a list of 6 integer code numbers

2. Write appropriate notation for the first, third, and seventh elements of the following arrays:
- **a.** `int grades[20]`
- ***b.*** `float volts[10]`
- ***c.*** `float amps[16]`
- ***d.*** `int dist[15]`
- ***e.*** `float velocity[25]`
- ***f.*** `float time[100]`

3a. Write individual `scanf( )` statements that can be used to enter values into the first, third, and seventh elements of each of the arrays declared in Exercises 2a through 2f.
b. Write a `for` loop that can be used to enter values for the complete array declared in Exercise 2a.

4a. Write individual `printf( )` statements that can be used to print the values in the first, third, and seventh elements of each of the arrays declared in Exercises 2a through 2f.
b. Write a `for` loop that can be used to display values for the complete array declared in Exercise 2a.

5. List the elements that will be displayed by the following sections of code:

a.
```
for (m = 1; m <= 5; ++m)
   printf("%d ", a[j]);
```
b.
```
for (k = 1; k <= 5; k = k + 2)
   printf("%d ", a[k]);
```
c.
```
for (j = 3; j <= 10; ++j);
   printf("%d ", b[j]);
```
d.
```
for (k = 3; k <= 12; k = k + 3)
   printf("%d ", b[k]);
```
e.
```
for (i = 2; i < 11; i = i + 2)
   printf("%d ", c[i]);
```

6a. Write a C program to input the following values into an array named `volts`: 10.95, 16.32, 12.15, 8.22, 15.98, 26.22, 13.54, 6.45, 17.59. After the data have been entered, have your program output the values.

b. Repeat Exercise 6a, but after the data have been entered, have your program display them in the following form:

```
10.95  16.32  12.15
 8.22  15.98  26.22
13.54   6.45  17.59
```

7. Write a C program to input eight integer numbers into an array named `temp`. As each number is input, add the numbers into a total. After all numbers are input, display the numbers and their average.

8a. Write a C program to input 10 integer numbers into an array named `fmax` and determine the maximum value entered. Your program should contain only one loop and the maximum should be determined as array element values are being input. (*Hint:* Set the maximum equal to the first array element, which should be input before the loop used to input the remaining array values.)

b. Repeat Exercise 8a, keeping track of both the maximum element in the array and the index number for the maximum. After displaying the numbers, print these two messages:

```
The maximum value is: ___
  This is element number ___ in the list of numbers
```

Have your program display the correct values in place of the underlines in the messages.

c. Repeat Exercise 8b, but have your program locate the minimum of the data entered.

9a. Write a C program to input the following integer numbers into an array named `grades`: 89, 95, 72, 83, 99, 54, 86, 75, 92, 73, 79, 75, 82, 73. As each number is input, add the numbers to obtain a total. After all numbers are input and the total is obtained, calculate the average of the numbers and use the average to determine the deviation of each value from the average. Store each deviation in an array named `deviation`. Each deviation is obtained as the element value less the average of all the data. Have your program display each deviation alongside its corresponding element from the `grades` array,

b. Calculate the variance of the data used in Exercise 9a. The variance is obtained by squaring each individual deviation and dividing the sum of the squared deviations by the number of deviations.

10. Write a C program that specifies three one-dimensional arrays named `volts`, `current`, and `resist`. Each array should be capable of holding ten elements. Using a `for` loop, input values for the `current` and `resist` arrays. The entries in the `volts` array should be the product of the corresponding values in the `current` and `resist` arrays (thus, `volts[i] = current[i] * resist[i]`). After all the data have been entered, display the following output:

```
Voltage     Current     Resistance
-------     -------     ----------
```

Under each column heading display the appropriate value.

11a. Write a program that inputs ten float numbers into an array named `raw`. After ten user-input numbers are entered into the array, your program should cycle through `raw` ten times. During each pass through the array, your program should select the lowest value in `raw` and place the selected value in the next available slot in an array named `sorted`. Thus, when your program is complete, the `sorted` array should contain the numbers in `raw` in sorted order from lowest to highest. (*Hint:* Make sure to reset the lowest value selected during each pass to a very high number so that it is not selected again. You will need a second `for` loop within the first `for` loop to locate the minimum value for each pass.)

b. The method used in Exercise 11a to sort the values in the array is very inefficient. Can you determine why? What might be a better method of sorting the numbers in an array?

6.2 Array Initialization

Array elements can be initialized within their declaration statements in the same manner as scalar variables, except that the initializing elements must be included in braces. Examples of such initializations are:[2]

```
int grades[5] = {98, 87, 92, 79, 85};
char codes[6] = {'s', 'a', 'm', 'p', 'l', 'e'};
double width[7] = {10.96, 6.43, 2.58, .86, 5.89, 7.56, 8.22};
```

Initializers are applied in the order they are written, with the first value used to initialize element 0, the second value used to initialize element 1, and so on, until all values have been used. Thus, in the declaration

```
int grades[5] = {98, 87, 92, 79, 85};
```

`grades[0]` is initialized to `98`, `grades[1]` is initialized to `87`, `grades[2]` is initialized to `92`, `grades[3]` is initialized to `79`, and `grades[4]` is initialized to `85`.

Since white space is ignored in C, initializations may be continued across multiple lines. For example, the declaration

```
int gallons[20] = {19, 16, 14, 19, 20, 18,  /* initializing values */
                   12, 10, 22, 15, 18, 17,  /* may extend across  */
                   16, 14, 23, 19, 15, 18,  /* multiple lines     */
                   21, 5};
```

uses four lines to initialize all the array elements.

[2] In older versions of C (non-ANSI) these declarations must be preceded with the keyword `static` if the declaration is contained within `main( )` or any other C function. This is discussed in more detail in the next chapter.

If the number of initializers is less than the declared number of elements listed in square brackets, the initializers are applied starting with array element zero. Thus, in the declaration

```
float length[7] = {7.8, 6.4, 4.9, 11.2};
```

only `length[0]`, `length[1]`, `length[2]`, and `length[3]` are initialized with the listed values. The other array elements will be initialized to zero.

Unfortunately, there is no method of either indicating repetition of an initialization value or initializing later array elements without first specifying values for earlier elements.

A unique feature of initializers is that the size of an array may be omitted when initializing values are included in the declaration statement. For example, the declaration

```
int gallons[] = {16, 12, 10, 14, 11};
```

reserves enough storage room for five elements. Similarly, the following two declarations are equivalent:

```
char codes[6] = {'s', 'a', 'm', 'p', 'l', 'e'};
char codes[] = {'s', 'a', 'm', 'p', 'l', 'e'};
```

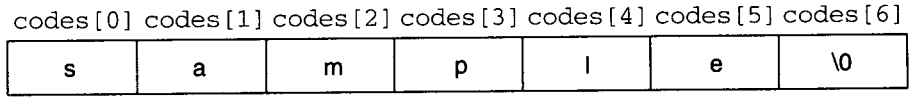
Both of these declarations set aside six character locations for an array named `codes`. An interesting and useful simplification can also be used when initializing character arrays. For example, the declaration

```
char codes[] = "sample";    /* no braces or commas */
```

uses the string `"sample"` to initialize the `codes` array. Recall that a string is any sequence of characters enclosed in double quotes. This last declaration creates an array named `codes` having seven elements and fills the array with the seven characters illustrated in Figure 6–6. The first six characters, as expected, consist of the letters s, a, m, p, l, and e. The last character, which is the escape sequence \0, is called the null character. The null character is automatically appended to all strings by the C compiler. This character has an internal storage code that is numerically equal to zero (the storage code for the zero character has a numerical value of decimal 48, so the two cannot be confused by the computer), and is used as a marker, or sentinel, to mark the end of a string. As we shall see in Chapter 11, this marker is invaluable when manipulating strings of characters.

Once values have been assigned to array elements, either through initialization within the declaration statement, using the interactive input described in Section 6.1, or by assignment the array elements can be processed as described in

FIGURE 6–6 A String Is Terminated with a Special Sentinel



the previous section. For example, Program 6-3 illustrates the initialization of array elements within the declaration of the array and then uses a `for` loop to locate the maximum value stored in the array.

Program 6-3

```
#include <stdio.h>
main( )
{
  int i, max, nums[5] = {2, 18, 1, 27, 16};

  max = nums[0];
  for (i = 1; i <= 4; ++i)
    if (max < nums[i])
      max = nums[i];

  printf("The maximum value is %d", max);
}
```

The output produced by Program 6-3 is:

```
The maximum value is 27
```

Exercises 6.2

1. Write array declarations, including initializers, for the following:
   a. a list of ten integer grades: 89, 75, 82, 93, 78, 95, 81, 88, 77, 82
   b. a list of five double precision amounts: 10.62, 13.98, 18.45, 12.68, 14.76
   c. a list of 100 double precision interest rates; the first six rates are 6.29, 6.95, 7.25, 7.35, 7.40, 7.42
   d. a list of 64 floating point temperatures; the first ten temperatures are 78.2, 69.6, 68.5, 83.9, 55.4, 67.0, 49.8, 58.3, 62.5, 71.6
   e. a list of 15 character codes; the first seven codes are f, j, m, q, t, w, z

2. Write an array declaration statement that stores the following values in an array named `volts`: 16.24, 18.98, 23.75, 16.29, 19.54, 14.22, 11.13, 15.39. Include these statements in a program that displays the values in the array.

3. Write a program that uses an array declaration statement to initialize the following numbers in an array named `slopes`: 17.24, 25.63, 5.94, 33.92, 3.71, 32.84, 35.93, 18.24, 6.92. Your program should locate and display both the maximum and minimum values in the array.

4. Write a program that stores the following prices in an array named `prices`: 9.92, 6.32, 12.63, 5.95, 10.29. Your program should also create two arrays named `units` and `amounts`, each capable of storing five double precision numbers. Using a `for` loop and a `scanf( )` function call, have your program accept five user-input numbers into the `units` array when the program is run. Your program should store the product of the corresponding values in the `prices` and `units` arrays in the `amounts` array (for example, `amounts[1] =`

prices[1] * units[1]) and display the following output (fill in the table appropriately):

```
Price     Units     Amount
-----     -----     ------
 9.92       .          .
 6.32       .          .
12.63       .          .
 5.95       .          .
10.29       .          .
                      ------
Total:                   .
```

5. The string of characters "Good Morning" is to be stored in a character array named goodstr1. Write the declaration for this array in three different ways.

6 a. Write declaration statements to store the string of characters "Input the Following Data" in a character array named messag1, the string "------------------------" in the array named messag2, the string "Enter the Date: " in the array named messag3, and the string "Enter the Account Number: " in the array named messag4.

b. Include the array declarations written in Exercise 6a in a program that uses the printf( ) function to display the messages. For example, the statement printf("%s", messag1); causes the string stored in the messag1 array to be displayed. Your program will require four such statements to display the four individual messages. Using the printf( ) function with the %s control sequence to display a string requires that the end-of-string marker \0 is present in the character array used to store the string.

7 a. Write a declaration to store the string "This is a test" into an array named strtest. Include the declaration in a program to display the message using the following loop:

```
for (i = 0; i <= 13; ++i)
  printf("%c", strtest[i]);
```

b. Modify the for statement in Exercise 7a to display only the array characters t, e, s, and t.

c. Include the array declaration written in Exercise 7a in a program that uses the printf( ) function to display characters in the array. For example, the statement printf("%s", strtest); will cause the string stored in the strtest array to be displayed. Using this statement requires that the last character in the array is the end-of-string marker \0.

d. Repeat Exercise 7a using a while loop. (*Hint:* Stop the loop when the \0 escape sequence is detected. The expression while (strtest[i] != '\0') can be used.)

6.3 Two-Dimensional Arrays

A *two-dimensional* array, which is also referred to as a table, consists of both rows and columns of elements. For example, the array of numbers

```
 8  16   9  52
 3  15  27   6
14  25   2  10
```

is called a two-dimensional array of integers. This array consists of three rows and four columns. To reserve storage for this array, both the number of rows and the number of columns must be included in the array's declaration. Calling the array `val`, the correct specification for this two-dimensional array is:

```
int val [3][4];
```

Similarly, the declarations

```
float volts [10][5];
char code [6][26];
```

declare that the array `volts` consists of 10 rows and 5 columns of floating point numbers and that the array `code` consists of 6 rows and 26 columns, with each element capable of holding one character.

In order to locate each element in a two-dimensional array, an element is identified by its position in the array. As illustrated in Figure 6–7, the term `val[1][3]` uniquely identifies the element in row 1, column 3. As with one-dimensional array variables, double-dimensional array variables can be used anywhere scalar variables are valid. Examples using elements of the `val` array are:

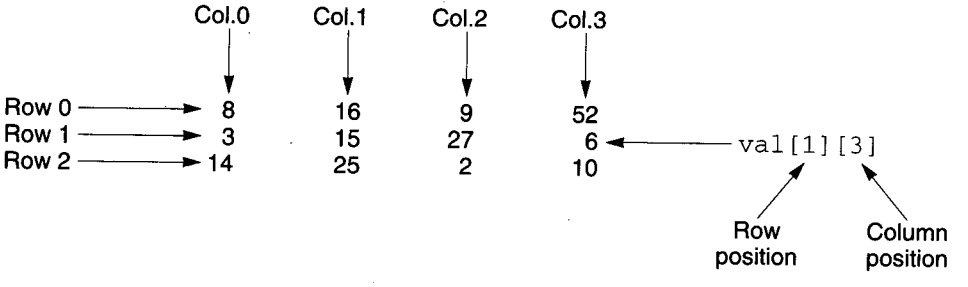
```
watts = val[2][3];
val[0][0] = 62;
newnum = 4 * (val[1][0] - 5);
sum_row = val[0][0] + val[0][1] + val[0][2] + val[0][3];
```

The last statement causes the values of the four elements in row 0 to be added and the sum to be stored in the scalar variable `sum_row`.

As with one-dimensional arrays, two-dimensional arrays can be initialized from within their declaration statements. This is done by listing the initial values within braces and separating them by commas. Additionally, braces can be used to separate individual rows. For example, the declaration

```
int val[3][4] = { {8,16,9,52},
                  {3,15,27,6},
                  {14,25,2,10} };
```

FIGURE 6–7 Each Array Element Is Identified by Its Row and Column Position



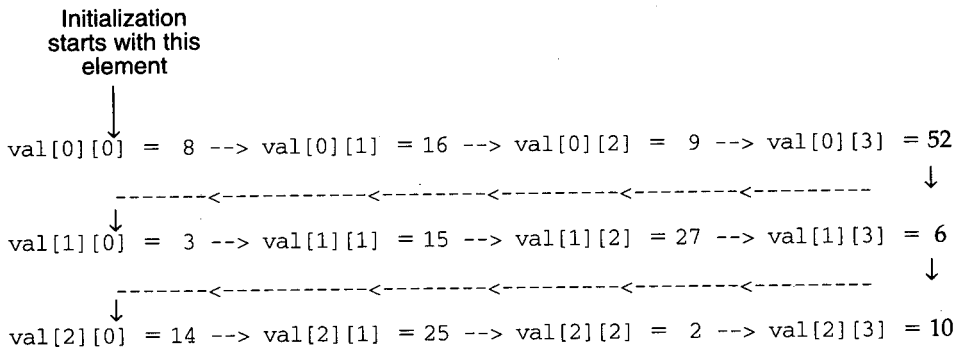


FIGURE 6–8 Storage and Initialization of the val[ ] array

declares `val` to be an array of integers with three rows and four columns, with the initial values given in the declaration. The first set of internal braces contains the values for row 0 of the array, the second set of internal braces contains the values for row 1, and the third set of braces the values for row 2.

Although the commas in the initialization braces are always required, the inner braces can be omitted. Thus, the initialization for `val` may be written as

```
int val[3][4] = {8,16,9,52,
                 3,15,27,6,
                 14,25,2,10};
```

The separation of initial values into rows in the declaration statement is not necessary since the compiler assigns values beginning with the `[0][0]` element and proceeds row by row to fill in the remaining values. Thus, the initialization

```
int val[3][4] = {8,16,9,52,3,15,27,6,14,25,2,10};
```

is equally valid but does not clearly illustrate to another programmer where one row ends and another begins.

As illustrated in Figure 6–8, the initialization of a two-dimensional array is done in row order. First, the elements of the first row are initialized, then the elements of the second row are initialized, and so on, until the initializations are completed. This row ordering is also the same ordering used to store two-dimensional arrays. That is, array element `[0][0]` is stored first, followed by element `[0][1]`, followed by element `[0][2]` and so on. Following the first row's elements are the second row's elements, and so on for all the rows in the array.

As with one-dimensional arrays, two-dimensional arrays may be displayed by individual element notation or by using loops (`while`, `for`, or `do`). This is illustrated by Program 6-4 (p. 256), which displays all the elements of a three-by-four two-dimensional array using two different techniques.

Following is the display produced by Program 6-4:

```
Display of val array by explicit element
 8  16   9  52
 3  15  27   6
14  25   2  10
```

Program 6-4

```
#include <stdio.h>
main( )
{
  int i, j, val[3][4] = {8,16,9,52,
                         3,15,27,6,
                         14,25,2,10};

  printf("\nDisplay of val array by explicit element");
  printf("\n%2d  %2d  %2d  %2d",val[0][0],val[0][1],val[0][2],val[0][3]);
  printf("\n%2d  %2d  %2d  %2d",val[1][0],val[1][1],val[1][2],val[1][3]);
  printf("\n%2d  %2d  %2d  %2d",val[2][0],val[2][1],val[2][2],val[2][3]);
  printf("\n\nDisplay of val array using a nested for loop");
  for (i = 0; i < 3; ++i)
  {
    printf("\n");      /* print a new line for each row */
    for (j = 0; j < 4; ++j)
      printf("%2d  ", val[i][j]);
  }
}
```

```
Display of val array using a nested for loop
 8  16   9  52
 3  15  27   6
14  25   2  10
```

The first display of the `val` array produced by Program 6-4 is constructed by explicitly designating each array element. The second display of array element values, which is identical to the first, is produced using a nested `for` loop. Nested loops are especially useful when dealing with two-dimensional arrays because they allow the programmer to easily designate and cycle through each element. In Program 6-4, the variable `i` controls the outer loop and the variable `j` controls the inner loop. Each pass through the outer loop corresponds to a single row, with the inner loop supplying the appropriate column elements. After a complete column is printed a new line is started for the next row. The effect is a display of the array in a row-by-row fashion.

Once two-dimensional array elements have been assigned array processing can begin. Typically, `for` loops are used to process two-dimensional arrays because, as previously noted, they allow the programmer to easily designate and cycle through each array element. For example, the nested `for` loop illustrated in Program 6-5 is used to multiply each element in the `val` array by the scalar number 10 and display the resulting value.

Following is the output produced by Program 6-5:

```
Display of multiplied elements

 80  160   90  520
 30  150  270   60
140  250   20  100
```

Program 6-5

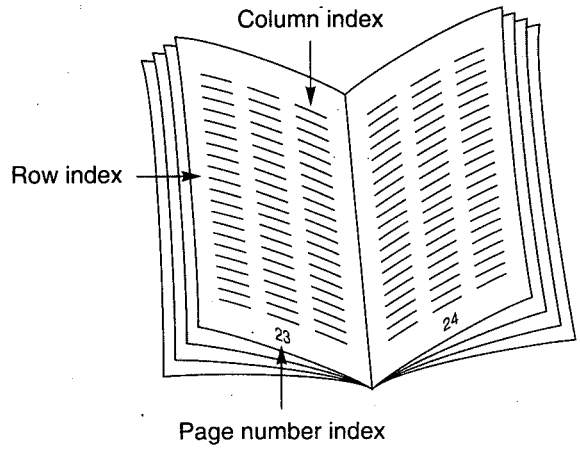
```
#include <stdio.h>
main( )
{
  int i, j, val[3][4] = {8,16,9,52,
                         3,15,27,6,
                         14,25,2,10};

  /* multiply each element by 10 and display it */
  printf("\n\nDisplay of multiplied elements\n");
  for (i = 0; i < 3; ++i)
  {
    printf("\n");        /* print a blank line */
    for (j = 0; j < 4; ++j)
    {
      val[i][j] = val[i][j] * 10;
      printf("%3d  ", val[i][j]);
    }  /* end of inner loop */
  }    /* end of outer loop */
}
```

Larger-Dimensional Arrays

Although arrays with more than two dimensions are not commonly used, C does allow any number of dimensions to be declared. This is done by listing the maximum size of all dimensions for the array. For example, the declaration `int response [4][10][6];` declares a three-dimensional array. The first element in the array is designated as `response [0][0][0]` and the last element as `response [3][9][5]`.

Conceptually, as illustrated in Figure 6–9, a three-dimensional array can be

FIGURE 6–9 Representation of a Three-Dimensional Array



viewed as a book of data tables. Using this visualization, the first subscript can be thought of as the location of the desired row in a table, the second subscript value as the desired column, and the third subscript value, which is often called the "rank," as the page number of the selected table.

Similarly, arrays of any dimension can be declared. Conceptually, a four-dimensional array can be represented as a shelf of books, where the fourth dimension is used to declare a desired book on the shelf, and a five-dimensional array can be viewed as a bookcase filled with books where the fifth dimension refers to a selected shelf in the bookcase. Using the same analogy, a six-dimensional array can be considered as a single row of bookcases where the sixth dimension references the desired bookcase in the row; a seven-dimensional array can be considered as multiple rows of bookcases where the seventh dimension references the desired row, and so on. Alternatively, arrays of three-, four-, five-, six-, etc. dimensional arrays can be viewed as mathematical *n*-tuples of order three, four, five, six, etc., respectively.

Exercises 6.3

1. Write appropriate specification statements for:
   a. an array of integers with 6 rows and 10 columns
   b. an array of integers with 2 rows and 5 columns
   c. an array of characters with 7 rows and 12 columns
   d. an array of characters with 15 rows and 7 columns
   e. an array of floating point numbers with 10 rows and 25 columns
   f. an array of floating point numbers with 16 rows and 8 columns

2. Determine the output produced by the following program:

```
#include <stdio.h>
main( ) {
  int i, j, val[3][4] = {8,16,9,52,3,15,27,6,14,25,2,10};

  for (i = 0; i < 3; ++i)
    for (j = 0; j < 4; ++j)
      printf("%2d  ", val[i][j]);
}
```

3 *a.* Write a C program that adds the values of all elements in the `val` array used in Exercise 2 and displays the total.
b. Modify the program written for Exercise 3a to display the total of each row separately.

4. Write a C program that adds equivalent elements of the two-dimensional arrays named `first` and `second`. Both arrays should have two rows and three columns. For example, element `[1][2]` of the resulting array should be the sum of `first[1][2]` and `second[1][2]`. The first and second arrays should be initialized as follows:

FIRST			SECOND		
16	18	23	24	52	77
54	91	11	16	19	59

5 *a.* Write a C program that finds and displays the maximum value in a two-dimensional

array of integers. The array should be declared as a four-by-five array of integers and initialized with these data:

```
16,22,99,4,18,-258,4,101,5,98,105,6,15,2,45,33,88,72,16,3
```

b. Modify the program written in Exercise 5a so that it also displays the maximum value's row and column subscript numbers.

6. Write a C program to select the values in a four-by-five array of integers in increasing order and store the selected values in the single-dimensional array named `sort`. Use the data given in Exercise 5a to initialize the two-dimensional array.

7 a. A professor has constructed a two-dimensional array of float numbers having 3 rows and 5 columns. This array currently contains the test grades of the students in the professor's advanced compiler design class. Write a C program that reads 15 array values and then determines the total number of grades in the ranges less than 60, greater than or equal to 60 and less than 70, greater than or equal to 70 and less than 80, greater than or equal to 80 and less than 90, and greater than or equal to 90.
b. Entering 15 grades each time the program written for Exercise 7a is run is cumbersome. What method is appropriate for initializing the array during the testing phase?
c. How might the program you wrote for Exercise 7a be modified to include the case of no grade being present? That is, what grade could be used to indicate an invalid grade and how would your program have to be modified to exclude counting such a grade?

6.4 Applications

Arrays are extremely useful for plotting data on either a video screen or a standard line printer. In this section we present a simple but elegant method of constructing such plots. The first application presents the basic method and uses it to produce modest plots. The second application incorporates data scaling to ensure that the plot fits within the area of the video screen or paper, regardless of the range of data plotted.

Application 1: Curve Plotting

In graphing data on either a video screen or printer two basic constraints must be considered. The first constraint is that both devices automatically move in a forward direction, which means that our graphs should avoid the need to "back up" (although there are methods for reversing the cursor motion on a video screen, all of our programs will be constructed to work for both printers and screens). The second constraint is that both printer paper and video displays are restricted in the horizontal direction to displaying a maximum of either 80 or 132 characters. No such restriction exists in the vertical direction because the paper length is effectively unlimited and the video display scrolls forward. For this reason our plots will always be constructed "sideways" with the y axis horizontal and the x axis vertical. With these two constraints in mind, consider the plot shown in Figure 6–10 (p. 260).

As illustrated in Figure 6–10, the graph is plotted sideways with the y axis displayed across the top of the graph and the x axis displayed down the side. Omitting, for the moment, the two header lines,

```
                                y axis
+-------------------------------------------------------->
```

the actual graph of the data points consists of 15 individual lines, as follows:

```
line 1:  |                                                *
line 2:  |                                    *
line 3:  |                              *
line 4:  |                        *
line 5:  |                  *
line 6:  |            *
line 7:  |      *
line 8:  |   *
line 9:  |     *
line 10: |       *
line 11: |             *
line 12: |                       *
line 13: |                                 *
line 14: |                                      *
line 15: |                                                *
```

Notice that individually, each line consists of only two printed symbols, a bar (|) and an asterisk (*). The bar is always displayed in column 1 and the asterisk is positioned to indicate an appropriate y value. With these points in mind, it is rather easy to construct these 15 lines. To do this we will first construct an exact image of the first line to be printed in an array of characters. After the array is constructed and printed it will be used to construct an image of the second line. After the second image is displayed the same array is used to construct an image of the third line, and so on until all 15 lines have been displayed. To make sure that the elements in the array can be displayed on a page the array should be

FIGURE 6–10

```
                                y axis
+-------------------------------------------------------->
|                                                      *
|                                              *
|                                    *
|                          *
|                    *
|              *
|         *
|      *
|    *
|   *
|        *
|              *
|                    *
|                          *
|                                    *
|                                              *
|                                                      *
```

smaller than the maximum horizontal width of either the paper or video screen being used for the display.

As illustrated in Figure 6–11, the array, called `line`, is filled with blanks, except for the first element, which stores the bar symbol, and one other element, which stores an asterisk.

Using the `line` array to store the image of each line before it is printed, our graphing approach is:

Step 1. Store an asterisk in the desired array element.
Step 2. Print the array.
Step 3. Reset the element to a blank.
Step 4. Repeat Steps 1 through 3 until the required number of lines have been displayed.

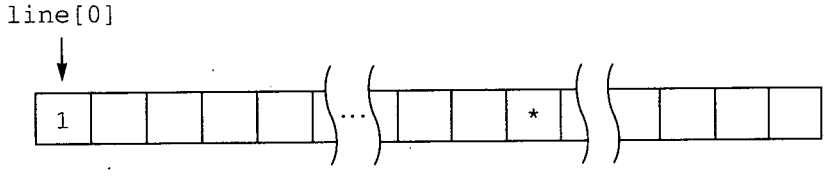
These four steps are easily implemented using a `for` loop having the form:

```
for (x = 1; x <= 15; ++x)
{
  calculate a value for y
  line[y] = '*';      /* set character to an asterisk */
  printf("\n%s",line);
  line[y] = ' ';      /* reset character to a blank */
}
```

The calculation of the y value, which is then used as a subscript for the `line` array, depends on the graph being plotted. For the graph illustrated in Figure 6–10, the equation $y = (x-8)^2 + 3$ was used.[3] Incorporating this into the `for` loop yields this executable code:

```
for (x = 1; x <= 15; ++x)
{
  y = pow((x-8),2.0) + 3;
  line[y] = '*';      /* set character to an asterisk */
  printf("\n%s",line);
  line[y] = ' ';      /* reset character to a blank */
}
```

FIGURE 6–11 The `line` Array



[3] To use the y value as a subscript for the line array requires that this value be an integer between the numbers zero and 72, inclusive. The curve $y = (x-8)^2 + 3$ was selected precisely because it yielded y values within this range. In the next application an algorithm is presented for scaling any y values into the required range.

Program 6-6 includes this code within a working program.

Program 6-6

```
#include <stdio.h>
#include <math.h>
main( )
{
  int x, y;
  char line[ ]="|                                                    ";

  for (x = 1; x <= 15; ++x)
  {
    y = pow((x-8),2.0) + 3;
    line[y] = '*';      /* set character to an asterisk */
    printf("\n%s",line);
    line[y] = ' ';      /* reset character to a blank */
  }
}
```

Notice in Program 6-6 that the `line` array is declared and initialized with a bar symbol and the remaining elements with blanks. The `for` loop then calculates a *y* value, uses this value as a subscript to locate where the asterisk should be placed in the `line` array, displays the array, and restores a blank in place of the asterisk. This process is repeated 15 times, resulting in the plot illustrated in Figure 6–12.

FIGURE 6–12 The Display Produced by Program 6-6

```
|                                                    *
|                                              *
|                                    *
|                          *
|                    *
|              *
|         *
|    *
|  *
|    *
|         *
|              *
|                    *
|                          *
|                                    *
|                                              *
```

In reviewing Program 6-6 two observations must be made. First, a y axis has not been explicitly included on the output. This is a minor omission that can be rectified by Program 6-7.

Program 6-7

```
#include <stdio.h>
#include <math.h>
main( )
{
  int x,y;
  char label[ ]="                    y axis";
  char axis[ ]="+-------------------------------------------------------->";
  char line[ ]="|                                                        ";

  printf("\n%s",label);
  printf("\n%s",axis);
  for (x = 1; x <= 15; ++x)
  {
    y = pow((x-8),2.0) + 3;
    line[y] = '*';      /* set character to an asterisk */
    printf("\n%s",line);
    line[y] = ' ';      /* reset character to a blank */
  }
}
```

Notice that Program 6-7 is essentially the same as Program 6-6 with the addition of two array declarations for the y axis and two `printf( )` function calls to display the `label` and y axis strings. These `printf( )` calls are placed before the `for` loop to display the header lines. Program 6-7 produces the completed plot shown in Figure 6–13.

A more serious problem with both Programs 6-6 and 6-7 is that negative y values and y values greater than the width of the `line` array cannot be accommodated as a subscript to the `line` array. To accommodate graphs with such values requires scaling of the y values to fit within the `line` array's subscript range. The scaling algorithm to do this, which ensures that our plotting program works for any y value, is presented in the next application.

Application 2: Data Scaling

A common problem encountered in plotting data is the need to scale values to fit within the width of the paper or video screen before a plotting routine can be used. Equation 1 provides the required scaling formula

$$\text{Scaled value} = \frac{\text{Original value} - \text{Minimum value}}{\text{Maximum value} - \text{Minimum value}} x \qquad (W-1)$$

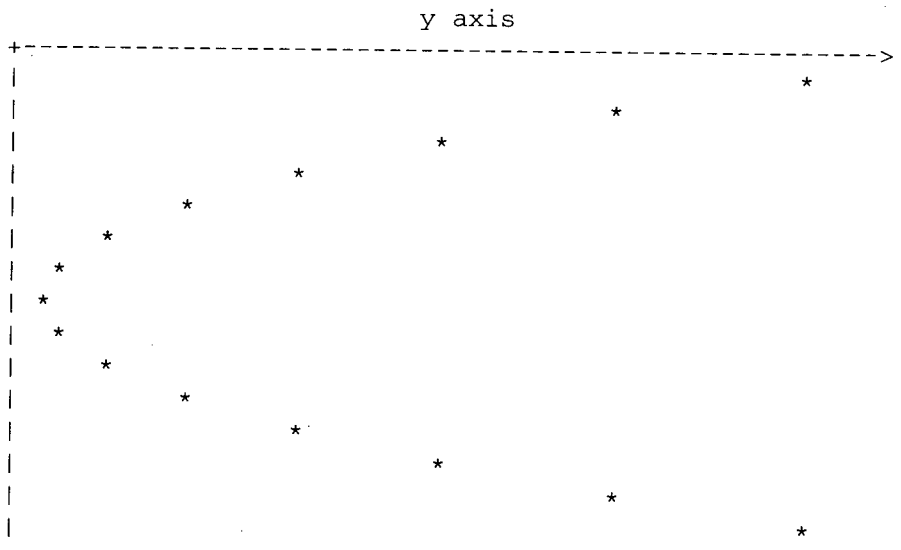


FIGURE 6–13

where the Maximum and Minimum values are the respective maximum and minimum values for the complete set of data values being plotted, and W is the desired width of the paper or video display. The term

$$\frac{\text{Original value} - \text{Minimum value}}{\text{Maximum value} - \text{Minimum value}}$$

in Equation 1 forces each original y value to lie within the range 0 to 1, with the minimum data value corresponding to 0 and the maximum data value to 1. Multiplying this result by the term (W-1) produces values between 0 and (W-1), for a total width of W.

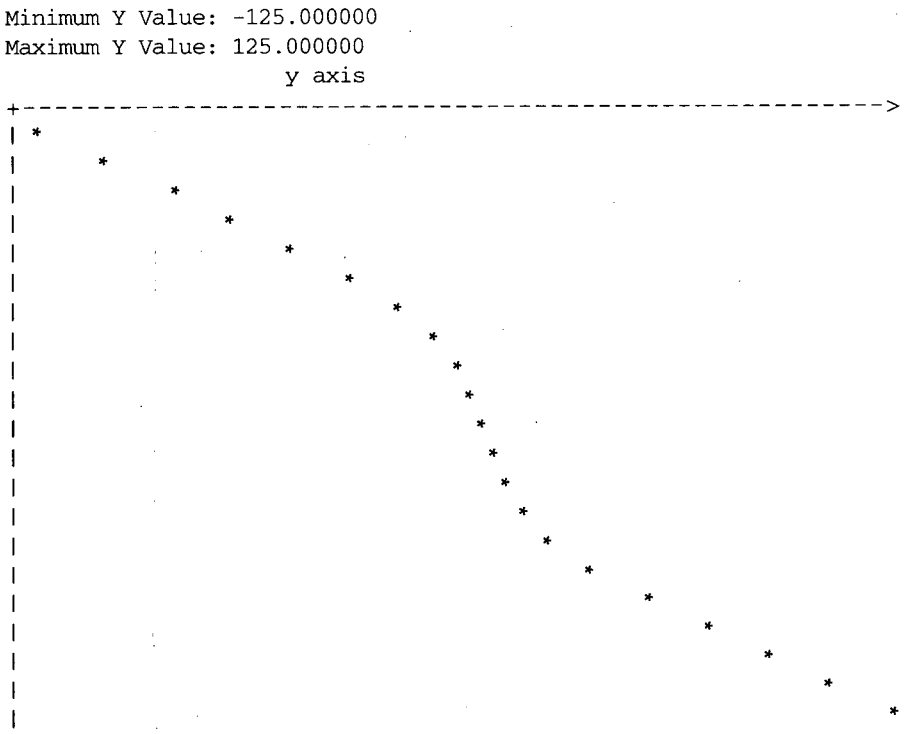
For example, the second column in Table 6–1 lists y values of the equation $y = -x^3$ for values of x between -5 and 5, in increments of 0.5. As shown in column 2, the maximum and minimum y values are $+125$ and -125, respectively.

For purposes of illustration assume that the width of the display area for plotting each y value in column 2 is 55 characters wide. Also assume that a single character of this display area is to be used for an axis symbol, such as the bar (|). This leaves a total width, W, of 54 for the actual data display. Applying Equation 1 to the data of column 2 with $W = 54$, and using the correct minimum and maximum values of -125 and 125, respectively, yields the values listed in column 3 of the table. Notice that the minimum value of -125 is converted to the value 0.000 and the maximum value of 125 is converted to the value 53.000. The last column in the table gives the rounded and integerized values of the scaled numbers listed in the third column. Notice that the values in column 4 range from 0 to 53, for a total range of 54 possible y values. These values can be used directly by the curve plotting routine presented in the previous application to create a graph similar to that shown in Figure 6–14.

TABLE 6–1 Values of the Equation $y = -x^3$

x	y	Scaled y	Rounded
−5.0	−125.000	0.000	0
−4.5	−91.125	7.186	7
−4.0	−64.000	12.932	13
−3.5	−42.875	17.411	17
−3.0	−27.000	20.776	21
−2.5	−15.625	23.188	23
−2.0	−8.000	24.804	25
−1.5	−3.375	25.786	26
−1.0	−1.000	26.288	26
−0.5	−0.125	26.473	26
0.0	0.000	26.500	27
0.5	0.125	26.527	27
1.0	1.000	26.712	27
1.5	3.375	27.216	27
2.0	8.000	28.196	28
2.5	15.625	29.813	30
3.0	27.000	32.224	32
3.5	42.875	35.590	36
4.0	64.000	40.068	40
4.5	91.125	45.819	46
5.0	125.000	53.000	53

FIGURE 6–14



Program 6-8, which was used to create Figure 6–14, includes the data scaling algorithm and the curve plotting routine used in Program 6-7.

Program 6-8

```
#include <stdio.h>
#include <math.h>
main( )
{
  int i, npts, nval[100];
  float x, xinc, fval, ymin, ymax, width, sval[100];
  char label[ ]="                                y axis";
  char axis[ ]="+-------------------------------------------------------->";
  char line[ ]="|                                                        ";

  ymax = 1.0e-5;
  ymin = 1.0e+5;
  x = -5.0;
  xinc = 0.5;
  npts = 21;
  width = 53;
  /* load up the data to be plotted and find the max and min values */
  for (i = 1; i <= npts; ++i)
  {
    sval[i] = pow(x,3.0);
    if (sval[i] > ymax)
      ymax = sval[i];
    if (sval[i] < ymin)
      ymin = sval[i];
    x = x + xinc;
  }
  /* scale all y values to be plotted */
  for (i = 1; i <= npts; ++i)
  {
    fval = ( (sval[i] - ymin)/(ymax - ymin) ) * width;
    nval[i] = fval + 0.5;    /* convert to an integer value */
  }
  /* produce the plot */
  printf("\nMinimum Y Value: %f",ymin);
  printf("\nMaximum Y Value: %f",ymax);
  printf("\n%s",label);
  printf("\n%s",axis);
  for (i = 1; i <= npts; ++i)
  {
    line[nval[i] + 2] = '*';      /* set character to an asterisk */
    printf("\n%s",line);
    line[nval[i] + 2] = ' ';      /* reset character to a blank */
  }
}
```

Additional Exercises for Chapter 6

1. Enter and run Program 6-7 on your computer system.

2. Modify Program 6-7 to plot the curve $y = x^3 - 4x^2 + 3x + 2$ for x equal to 0, 1, 2, 3, 4, 5, and 6.

3. Modify Program 6-7 to plot the curve $y = x^3 - 4x^2 + 3x + 2$ for x between -8 and $+8$, in increments of 0.5.

4. Modify Program 6-7 to plot the curve $y = 4x^3 - x^4$ for x between -10 and $+10$, in increments of 0.5.

5. When the switch illustrated in Figure 6–15 is closed at time $t = 0$, the voltage, V, across the capacitor is given by the equation $V = E(1 - e^{t/(RC)})$, where E is the voltage of the battery, R is the circuit resistance, C is the value of the capacitance, and t is in seconds. Assuming that $E = 50$, $R = 4000$, and $C = .005$, modify Program 6-7 to plot the voltage across the capacitor from $t = 0$ to $t = 30$, in increments of 1 second.

6. Enter and run Program 6-8 on your computer system.

7. Modify Program 6-8 to plot the voltage across the capacitor illustrated in Figure 6–15 from $t = 0$ to $t = 30$ seconds in increments of 1 second. For this problem assume that $E = 100$ volts, $R = 4000$, and $C = .005$.

8. Figure 6–16 illustrates a *harmonic oscillator*, which consists of an object of mass M

FIGURE 6–15 A Simple RC Circuit

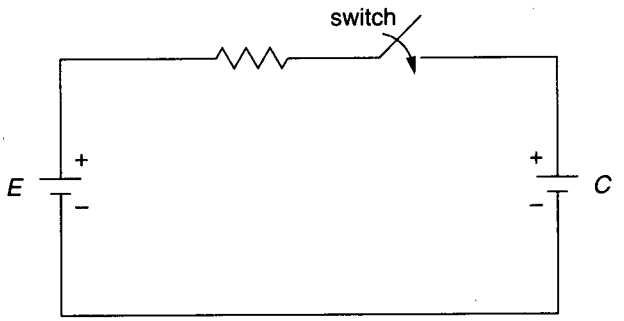
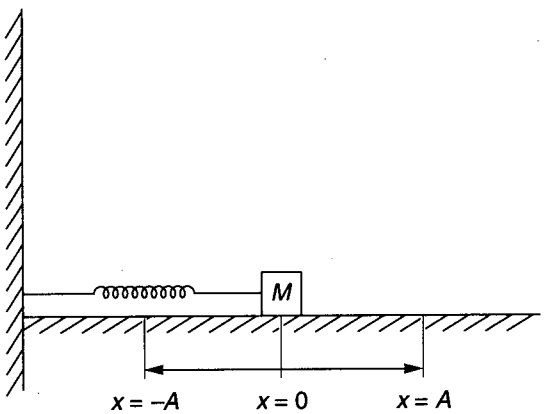


FIGURE 6–16 A Harmonic Oscillator



fastened to one end of a spring. The other end of the spring is attached to a wall, and the object is free to slide over a frictionless surface. Assuming the object is initially at rest (that is, the spring is neither stretched or compressed) and then pulled to position A at time $t = 0$, the position of the mass at any other time, t, is described by the equation $x = A \cos\left(\sqrt{k/m}\ t\right)$ where k is the spring constant in newtons/meter, m is the mass in units of kilograms, and A is the initial displacement in units of centimeters. Assuming A is 10 centimeters, k is 2000 newtons/meter, and m is 200 kilograms, modify Program 6-8 to plot the displacement of the mass from $t = 0$ to $t = 60$ seconds in increments of 1 second.

6.5 Common Programming Errors

Four common errors are associated with using arrays:

1. Forgetting to declare the array. This error results in a compiler error message equivalent to "invalid indirection" each time a subscripted variable is encountered within a program.
2. Using a subscript that references a nonexistent array element. For example, declaring the array to be of size 20 and using a subscript value of 25. This error is not detected by most C compilers. It can, however, result in a run-time error that results in a program "crash" or a value that has no relation to the intended array element. In either case this is usually an extremely troublesome error to locate. The only solution to this problem is to make sure, either by specific programming statements or by careful coding, that each subscript references a valid array element.
3. Not using a large enough conditional value in a `for` loop counter to cycle through all the array elements. This error usually occurs when an array is initially specified to be of size `n` and there is a `for` loop within the program of the form `for (i = 0; i < n; ++i)`. The array size is then expanded but the programmer forgets to change the interior for loop parameters.
4. Forgetting to initialize the array. Although many compilers automatically set all elements of integer and real valued arrays to zero and all elements of character arrays to blanks, it is up to the programmer to ensure that each array is correctly initialized before processing of array elements begins.

6.6 Chapter Summary

1. A one-dimensional array is a data structure that can be used to store a list of values of the same data type. Such arrays must be declared by giving the data type of the values that are stored in the array and the array size. For example, the declaration

```
int num[100];
```

creates an array of 100 integers.

2. Array elements are stored in contiguous locations in memory and referenced using the array name and a subscript, for example, `num[22]`. Any nonnegative integer-value expression can be used as a subscript and the subscript 0 always refers to the first element in an array.
3. Two-dimensional arrays are declared by specifying both a row and a column size. For example, the declaration

```
float rates[12][20];
```

reserves memory space for a table of 12 by 20 floating point values. Individual elements in a two-dimensional array are identified by providing both a row and a column subscript. The element in the first row and first column has row and column subscripts of 0.

6.7 Enrichment Study: Sorting Methods

Most programmers encounter the need to sort a list of data items at some time in their programming careers. For example, experimental results might have to be arranged in either increasing (ascending) or decreasing (descending) order for statistical analysis, lists of names may have to be sorted in alphabetical order, or a list of dates may have to be rearranged in ascending date order.

For sorting data, two major categories of sorting techniques exist, called internal and external sorts, respectively. *Internal sorts* are used when the data list is not too large and the complete list can be stored within the computer's memory, usually in an array. *External sorts* are used for much larger data sets that are stored in large external disk or tape files and cannot be accommodated within the computer's memory as a complete unit.

In this section we present two common internal sorts, the selection and exchange sorts. Although the exchange sort, also known as a "bubble sort," is the more common of the two, we will see that the selection sort is easier and frequently more efficient.

Selection Sort

In a selection sort the smallest (or largest) value is initially selected from the complete list of data and exchanged with the first element in the list. After this first selection and exchange, the next smallest (or largest) element in the revised list is selected and exchanged with the second element in the list. Since the smallest element is already in the first position in the list, this second pass need only consider the second through last elements. For a list consisting of n elements this process is repeated $n-1$ times, with each pass through the list requiring one less comparison than the previous pass.

For example, consider the list of numbers illustrated in Figure 6–17 (p. 270). The first pass through the initial list results in the number 32 being selected and exchanged with the first element in the list. The second pass, made on the reordered list, results in the number 155 being selected from the second through fifth elements. This value is then exchanged with the second element in the list. The third pass selects the number 307 from the third through fifth elements in the

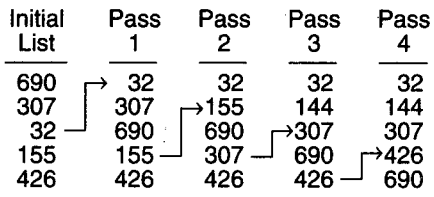


FIGURE 6–17 A Sample Selection Sort

Program 6-9

```
#include <stdio.h>
main( )
{
  int nums[10] = {22,5,67,98,45,32,101,99,73,10};
  int i, j, temp, moves, min, minind;

  moves = 0;
  for ( i = 0; i < 9; ++i)
  {
    min = nums[i];
    minind = i;
    for ( j = i + 1; j < 10; ++j)
      if (nums[j] < min)
      {
        min = nums[j];
        minind = j;
      }
    /* perform the switch */
    if (min < nums[i])
    {
      temp = nums[i];
      nums[i] = min;
      nums[minind] = temp;
      ++moves;
    }
  }
  printf("The sorted list, in ascending order, is:\n");
  for (i = 0; i < 10; ++i)
    printf("%d  ",nums[i]);
  printf("\n %d moves were made to sort this list\n", moves);
}
```

list and exchanges this value with the third element. Finally, the fourth and last pass through the list selects the remaining minimum value and exchanges it with the fourth list element. Although each pass in this example resulted in an exchange, no exchange would have been made in a pass if the smallest value were already in the correct location.

Program 6-9 implements a selection sort for a list of ten numbers that are stored in an array named nums.[4] For later comparison to an exchange sort, the number of actual moves made by the program to get the data into sorted order is counted and displayed.

Program 6-9 uses a nested `for` loop to perform the selection sort. The outer `for` loop causes nine passes to be made through the data, which is one less than the total number of data items in the list. For each pass the variable `min` is initially assigned the value `nums[i]`, where `i` is the outer `for` loop's counter variable. Since `i` begins at 0 and ends at 8, each element in the list is successively designated as the next exchange element.

The inner loop is used in Program 6-9 to cycle through the elements below the designated exchange element to select the next smallest value. Thus, this loop begins at the index value `i+1` and continues through the end of the list. When a new minimum is found its value and position in the list are stored in the variables named `min` and `minind`, respectively. Upon completion of the inner loop an exchange is made only if a value less than that in the designated exchange position was found.

Following is the output produced by Program 6-9:

```
The sorted list, in ascending order, is:
5  10  22  32  45  67  73  98  99  101
 8 moves were made to sort this list
```

Clearly the number of moves displayed depends on the initial order of the values in the list. An advantage of the selection sort is that the maximum number of moves that must be made is n-1, where n is the number of items in the list. Further, each move is a final move that results in an element residing in its final location in the sorted list.

A disadvantage of the selection sort is that $n(n\text{-}1)/2$ comparisons are always required, regardless of the initial arrangement of the data. This number of comparisons is obtained as follows: The last pass always requires one comparison, the next-to-last pass requires two comparisons, and so on to the first pass, which requires n-1 comparisons. Thus, the total number of comparisons is:

$$1 + 2 + 3 + \ldots n\text{-}1 = n(n\text{-}1)/2$$

Exchange Sort

In an exchange sort successive values in the list are compared, beginning with the first two elements. If the list is to be sorted in ascending (from smallest to largest) order, the smaller value of the two being compared is always placed

[4] If a non-ANSI compiler is used, the word `static` must be placed before the word `int` in the declaration statement for `nums[10]`.

before the larger value. For lists sorted in descending (from largest to smallest) order, the smaller of the two values being compared is always placed after the larger value.

For example, assuming that a list of values is to be sorted in ascending order, if the first element in the list is larger than the second, the two elements are interchanged. Then the second and third elements are compared. Again, if the second element is larger than the third, these two elements are interchanged. This process continues until the last two elements have been compared and exchanged, if necessary. If no exchanges were made during this initial pass through the data, the data are in the correct order and the process is finished; otherwise a second pass is made through the data, starting from the first element and stopping at the next-to-last element. The reason for stopping at the next-to-last element on the second pass is that the first pass always results in the most positive value "sinking" to the bottom of the list.

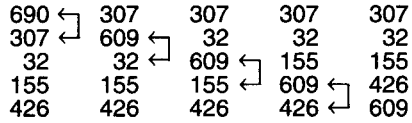
As a specific example of this process, consider the list of numbers illustrated in Figure 6–18. The first comparison results in the interchange of the first two element values, 690 and 307. The next comparison, between elements two and three in the revised list, results in the interchange of values between the second and third elements, 609 and 32. This comparison and possible switching of adjacent values is continued until the last two elements have been compared and possibly switched. This process completes the first pass through the data and results in the largest number moving to the bottom of the list. As the largest value sinks to its resting place at the bottom of the list, the smaller elements slowly rise or "bubble" to the top of the list. This bubbling effect of the smaller elements gave rise to the name "bubble sort" for this sorting algorithm.

As the first pass through the list ensures that the largest value always moves to the bottom of the list, the second pass stops at the next-to-last element. This process continues with each pass stopping at one higher element than the previous pass, until either n-1 passes through the list have been completed or no exchanges are necessary in any single pass. In both cases the resulting list is in sorted order.

Program 6-10 implements an exchange sort for the same list of ten numbers used in Program 6-9. For comparison to the earlier selection sort, the number of adjacent moves (exchanges) made by the program is also counted and displayed.[5]

As illustrated in Program 6-10, the exchange sort requires a nested loop. The outer loop in Program 6-10 is a `while` loop that checks if any exchanges were made in the last pass. It is the inner `for` loop that does the actual comparison and exchanging of adjacent element values.

FIGURE 6–18 The First Pass of an Exchange Sort



[5] If a non-ANSI compiler is used, the word `static` must be placed before the word `int` in the declaration statement for `nums[10]`.

Program 6-10

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
main( )
{
  static int nums[10] = {22,5,67,98,45,32,101,99,73,10};
  int i, temp, moves, npts, outord;

  moves = 0;
  npts = 10;
  outord = TRUE;
  while (outord && npts > 0)
  {
    outord = FALSE;
    for ( i = 0; i < npts - 1; ++i)
      if (nums[i] > nums[i+1])
      {
        temp = nums[i+1];
        nums[i+1] = nums[i];
        nums[i] = temp;
        outord = TRUE;
        ++moves;
      }
    --npts;
    }
  printf("The sorted list, in ascending order, is:\n");
  for (i = 0; i < 10; ++i)
    printf("%d  ",nums[i]);
  printf("\n %d moves were made to sort this list\n", moves);
}
```

Immediately before the inner loop's `for` statement is encountered the value of the variable `outord` is set to `TRUE` to indicate that the list is initially out of order (not sorted) and force the first pass through the list. If the inner loop then detects an element is out of order `outord` is again set to `TRUE`, which indicates that the list is still unsorted. The `outord` variable is then used by the outer loop to determine whether another pass through the data is to be made. Thus, the sort is stopped either because `outord` is `FALSE` after at least one pass has been completed or n-1 passes through the data have been made. In both cases the resulting list is in sorted order.

Following is the output produced by Program 6-10:

```
The sorted list, in ascending order, is:
5  10  22  32  45  67  73  98  99  101
 18 moves were made to sort this list
```

As with the selection sort, the number of moves required by an exchange sort depends on the initial order of the values in the list.

An advantage of the exchange sort is that processing is terminated whenever a sorted list is encountered. In the best case, when the data is in sorted order to begin with, an exchange sort requires no moves (the same for the selection sort) and only n-1 comparisons (the selection sort always requires $n(n\text{-}1)/2$ comparisons). In the worst case, when the data is in reverse sorted order, the selection sort does better. Here both sorts require $n(n\text{-}1)/2$ comparisons but the selection sort needs only n-1 moves while the exchange sort needs $n(n\text{-}1)/2$ moves. The additional moves required by the exchange sort result from the intermediate exchanges between adjacent elements to "settle" each element into its final position. In this regard the selection sort is superior because no intermediate moves are necessary. For random data, such as that used in Programs 6-9 and 6-10, the selection sort generally performs equal to or better than the exchange sort. For large numbers of data values there are more efficient sorting routines, such as the quick sort, which is of order $n\log_{10}n$ comparisons.

Writing Your Own Functions

Chapter Seven

7.1 Function Definitions and Declarations
7.2 Arrays as Arguments
7.3 Applications
7.4 Variable Scope
7.5 Variable Storage Class
7.6 Common Programming Errors
7.7 Chapter Summary
7.8 Enrichment Study: Programming Costs

As we have seen, each C program must contain a `main( )` function. The `main( )` function can call any number of other functions, which in turn can also call other functions. In the `main( )` functions written so far, we have used the `printf( )` and `scanf( )` functions as well as many mathematical library functions, such as `pow( )`, `abs( )`, and `sqrt( )`. In this chapter we learn how to write our own C functions, pass data to them, process the passed data, and return a result.

7.1 Function Definitions and Declarations

The purpose of a C function, whether from a library or user-written, is to receive data, operate on the data, and directly return at most a single value.[1] In creating our own functions we must be concerned with both the function itself and how it interfaces with other functions. Before describing how user-written functions are defined and declared, however, let us briefly review what we know about calling and using system-provided functions.

As we have already seen with the `printf( )` and `scanf( )` functions, a function is called, or used, by giving the function's name and passing any data to it in the parentheses following the function name (see Figure 7–1).

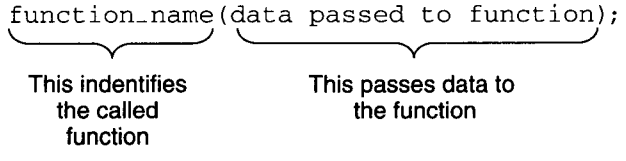
The called function must be able to accept the data passed to it by the function doing the calling. Only after the called function successfully receives the data can the data be manipulated to produce a useful result.

To clarify the process of sending and receiving data, consider Program 7-1, which calls a function named `find_max( )`. The program, as shown, is not yet complete. Once the function `find_max( )` is written and included in Program 7-1, the completed program, consisting of the functions `main( )` and `find_max( )`, can be run.

Let us examine declaring and calling the function `find_max( )` from the `main( )` function. We will then write `find_max( )` to accept the data passed to it and determine the largest or maximum value of the two passed values.

The function `find_max( )` is referred to as the *called function,* since it is called or summoned into action by its reference in the `main( )` function. The function that does the calling, in this case `main( )`, is referred to as the *calling function.* The terms called and calling come from standard telephone usage, where one party calls the other on a telephone. The party initiating the call is referred to as the calling party, and the party receiving the call is referred to as the called party. The same terms describe function calls. Within `main( )` the

FIGURE 7–1 Calling and Passing Data to a Function



[1] In Chapter 10 we will see how a function can indirectly return more than one value.

Program 7-1

```
#include <stdio.h>
main( )
{
  float firstnum, secnum, maxnum;
  float find_max(float, float);  /* the function prototype */

  printf("Enter a number: ");
  scanf("%f", &firstnum);
  printf("\nGreat! Please enter a second number: ");
  scanf("%f", &secnum);

  maxnum = find_max(firstnum,secnum);/*the function is called here*/
  printf("\nThe maximum of the two numbers entered is %f", maxnum);
}
```

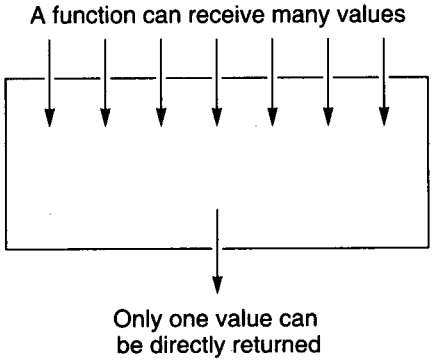
called function, in this case `find_max( )`, is declared as expecting to receive two floating point values and as returning one floating point value. This declaration is formally referred to as a *function prototype,* and is described in further detail later in this section. Let us now see how to write the function `find_max( )`.

Defining a Function

A function is defined when it is written. Each function is defined once (that is, written once) in a program and can then be used by any other function in the program that suitably declares it. In keeping with C's convention, a function is restricted to directly returning at most a single value (see Figure 7–2).

Like the `main( )` function, every C function consists of two parts, a function header and a function body, as illustrated in Figure 7–3. The purpose of the function header is to identify the data type of the value returned by the function, provide the function with a name, and specify the number, order, and type of argu-

FIGURE 7–2 A Function Returns at Most a Single Value



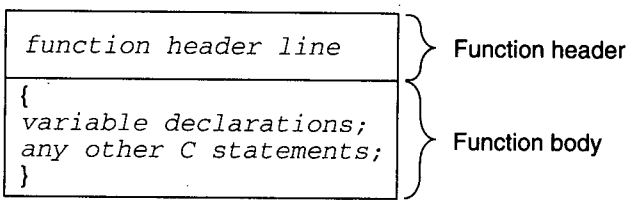


FIGURE 7–3 General Format of a Function

ments expected by the function. The purpose of the function body is to operate on the passed data and return, at most, one value back to the calling function. (We will see in Chapter 10 how a function can be made to return multiple values indirectly, using pointers.)

In ANSI C, a function header consists of a single line that contains the function's returned value type, its name, and the names and data types of its arguments. If the return value type is omitted, the function is defined to return an integer value by default. For example, the function header

```
float find_max(float x, float y) ← no semicolon
```

declares the returned data type and name of the function as well as declaring the names and data types of all arguments.[2] The argument names in the header line are formally referred to as *parameters* or *formal arguments*, and we shall use these terms interchangeably. The portion of the function header that contains the function name and parameters is formally referred to as a *function declarator*.

The function name and all parameter names in the header line, in this case `find_max`, `x`, and `y`, are chosen by the programmer. Any names selected according to the rules used to choose variable names can be used. All parameters listed in the function declarator must be separated by commas and must have their individual data types specified separately. If a data type is omitted, the parameter is of type integer by default. For example, the declarator `find_max(float x, y)` *does not* declare both of the parameters, `x` and `y`, to be of type `float`; rather, it declares `x` to be of type `float` and `y` to be of type `int`. Similarly, omitting the data type of the function immediately preceding the function's declarator defines the function's return value to be of type integer by default. Thus both function headers

```
int max_it(float x, float y)
```

and

```
max_it(float x, float y)
```

define the function `max_it( )` as returning an integer value.

---

[2] In non-ANSI C this single function header is written using these two lines:

```
float find_max(x, y)      /* header line           */
float x, y:               /* argument declarations */
```

Within a function header the keyword `void` is used to declare either that the function returns no value or that it has no arguments. For example, the function header

```
void display(int x, double y)
```

declares that the function `display( )` returns no value, while the function header

```
double print_message(void)
```

declares that the function `print_message( )` has no parameters but returns a value of type `double`. As illustrated, a function header line is never terminated with a semicolon.

Having written the function header for the `find_max( )` function as

```
float find_max(float x, float y)
```

we now construct the body of this function. For purposes of illustration, let us assume that the `find_max( )` function selects the larger of two numbers passed to it and returns this number to the calling routine.

As illustrated in Figure 7–4, a function body begins with an opening brace, `{`, contains any necessary variable declarations followed by any valid C statements, and ends with a closing brace, `}`. This should be familiar to you because it is the same structure used in all the `main( )` functions we have written.

In the body of the function `find_max( )` we will declare one variable to store the maximum of the two numbers passed to it. We will then use an `if-else` statement to find the maximum of the two numbers. Once the maximum value is determined, all that remains is to include a statement within the function to cause it to return this value to the calling function. To return a value a function must use a `return` statement, which has the form:

```
return(expression);
```

When the `return` statement is encountered, the expression inside the parentheses is evaluated first. The value of the expression is then automatically converted to the data type declared in the function header line before being sent back to the calling function. After the value is returned, program control reverts back to the calling function. Thus, the complete function definition for the `find_max( )` function is:

```
float find_max(float x, float y) /* function header           */
{                                /* start of function body    */
  float maxnum;                  /* variable declaration      */

  if (x >= y)                   /* find the maximum number    */
    maxnum = x;
  else
    maxnum = y;

  return(maxnum);                /* return the value          */
}                                /* end of function definition*/
```

```
{
  variable declarations (if any)
  other C statements
}
```

FIGURE 7–4 Structure of a Function Body

When this function is called, the parameter `x` will be used to store the first value passed to it and the parameter `y` will be used to store the second value passed at the time of the function call. The function itself will not know where the values come from when the call is made.

Note that within the `return` statement the data type of the returned variable correctly matched the data type in the function's header line. It is up to the programmer to ensure that this is so for every function returning a value. Failure to match the return value with the function's defined data type will not result in an error when your program is compiled, but it may lead to undesirable results since the returned value is always converted to the data type specified in the function's header line. Usually this is a problem only when the fractional part of a returned floating point or double precision number is truncated because the function was defined to return an integer value.

Having completely defined (written) the `find_max( )` function, let us now see how this function can be called.

Declaring a Function

Before a function can be called, it must be declared to the function that will do the calling. The declaration statement for a function is formally referred to as a *function prototype*. The function prototype tells the calling function the type of value that will be returned, if any, and the data type of the values that the calling function should transmit to the called function. For example, the function prototype previously used in Program 7-1,

```
float find_max(float, float);
```

declares that the function `find_max( )` expects two floating point values to be sent to it, and that this particular function returns a floating point value. Function prototypes may be placed with the variable declaration statements of the calling function, as in Program 7-1, or above the calling function name. Thus, the function prototype for `find_max( )` could have been placed either before or after the statement `#include <stdio.h>`, which contains the function prototypes for the `printf( )` and `scanf( )` functions called in `main( )`. (Reasons for the choice of placement are presented in Section 7.4.) The general form of function prototype statements is:

```
return-data-type function-name(list of argument data types);
```

The `data-type` refers to the data type that will be returned by the function and must match the data type used in the function's header line. Similarly, the list of argument data types must match those used in the function's definition. Further examples of function prototypes are:

```
int fmax(float, float);
float roi(int, char, char, double);
void display(double, double);
```

The function prototype for `fmax( )` declares that this function expects to receive two floating point arguments and will return an integer value. The function prototype for `roi( )` declares that this function requires four arguments consisting of an integer, two characters, and a double precision argument, in this order, and will return a floating point number. Finally, the function prototype for `display( )` declares that this function requires two double precision arguments and does not return any value. Such a function might be used to display the results of a computation directly, without returning any value to the called function.

The use of function prototypes permits error checking of parameter types by the compiler. If the function prototype does not agree with the return and parameter data types contained in the function's header line, an error message (typically TYPE MISMATCH) will occur. The prototype also serves another task: It ensures conversion of all arguments passed to the function to the declared argument data type when the function is called.

Calling a function is rather trivial. All that is required is that the name of the function be used and that any data passed to the function be enclosed within the parentheses following the function name. The items enclosed within the parentheses are called *actual arguments* of the called function (see Figure 7–5).

If a variable is one of the arguments in a function call, the called function receives a copy of the value stored in the variable. For example, the statement

```
maxnum = find_max(firstnum,secnum);
```

calls the function `find_max( )`, causes the values currently residing in the variables `firstnum` and `secnum` to be passed to `find_max( )`, and assigns the function's returned value to `maxnum`. The variable names in parentheses are actual arguments that provide values to the called function. After the values are passed, control is transferred to the called function.

As illustrated in Figure 7–6, the function `find_max( )` does not receive the variable names `firstnum` and `secnum` and has no knowledge of these variable names.[3] The function simply receives copies of the values in these variables and must itself determine where to store these values before it does anything else. Although this procedure for passing data to a function may seem surprising, it is

FIGURE 7–5 Calling and Passing Two Values to `find_max( )`

```
find_max (first num, secnum);
```

This calls the `find_max ( )` function

This causes two values to be passed to `find_max ( )`

[3] This is significantly different from computer languages such as FORTRAN, in which functions and subroutines receive access to the variables and can pass data back through them.

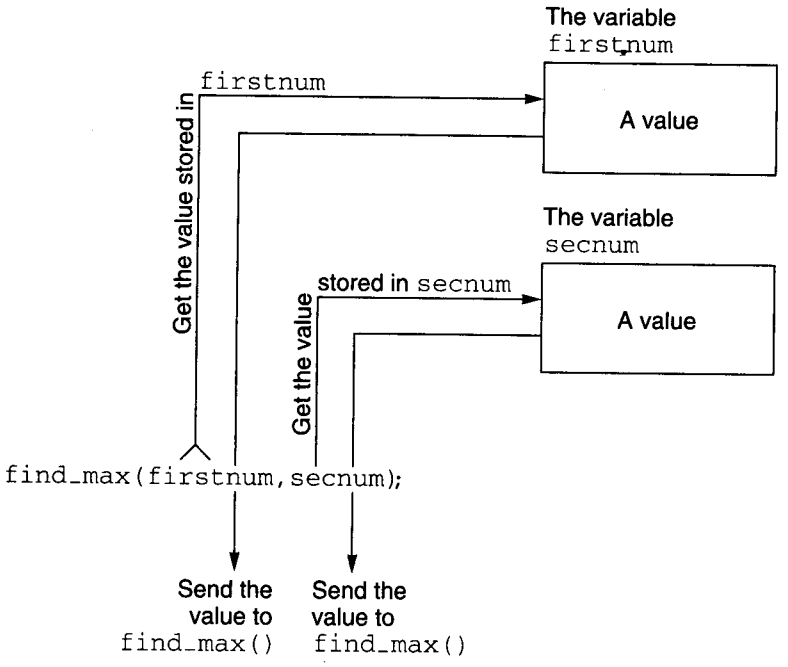
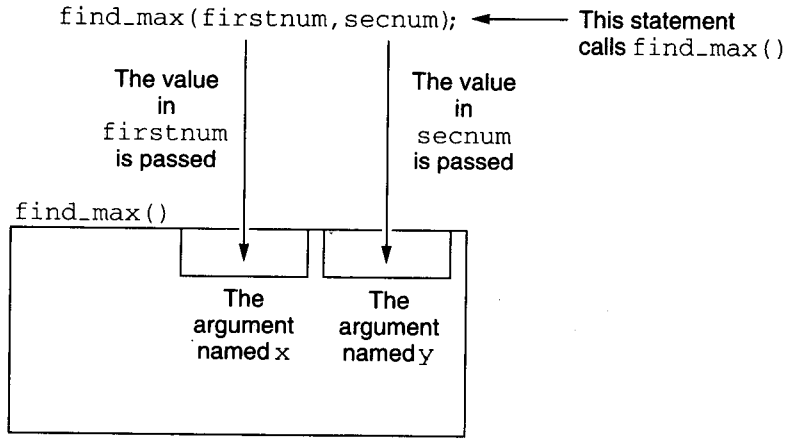


FIGURE 7–6 `find_max( )` Receives Actual Values

really a safety procedure for ensuring that a called function does not inadvertently change data stored in a variable. The function gets a copy of the data to use. It may change its copy and, of course, change any variables or arguments declared inside itself. However, unless specific steps are taken to do so, a function is not allowed to change the contents of variables declared in other functions.

The parameters in the function definition are used to store the values passed to the function when it is called. As illustrated in Figure 7–7, the parameters `x` and `y` of the `find_max( )` function are treated like variables by `find_max( )`,

FIGURE 7–7 Storing Values into Arguments



where the initialization of the values for these parameters occurs outside the function.

Program 7-2 includes the `find_max( )` function within the program code previously listed in Program 7-1.

Program 7–2

```
#include <stdio.h>
main( )
{
  float firstnum, secnum, maxnum;
  float find_max(float, float);  /* the function prototype */

  printf("Enter a number: ");
  scanf("%f", &firstnum);
  printf("\nGreat! Please enter a second number: ");
  scanf("%f", &secnum);

  maxnum = find_max(firstnum,secnum); /*the function is called here*/
  printf("\nThe maximum of the two numbers entered is %f", maxnum);
}

/* the following is the function find_max  */

float find_max(float x, float y) /* function header              */
{                                /* start of function body       */
  float maxnum;                  /* variable declaration         */

  if (x >= y)                   /* find the maximum number      */
    maxnum = x;
  else
    maxnum = y;

  return(maxnum);                /* return the value            */
}                                /* end of function definition */
```

Program 7-2 can be used to select and print the maximum of any two floating numbers entered by the user. Following is a sample run using Program 7-2:

```
Enter a number: 25.0
Great! Please enter a second number: 5.0

The maximum of the two numbers is 25.000000
```

In reviewing Program 7-2 it is important to note the four items we have introduced in this section. The first item is the function prototype (declaration) of

find_max( ) within main( ). This statement, which ends with a semicolon as all declaration statements do, alerts main( ) to the data type that find_max( ) will be returning. The second item to notice in main( ) is the call find_max( ) and the use of an assignment statement to store the returned value in the variable maxnum. We have also made sure to correctly declare maxnum as a floating point variable within main( )'s variable declarations so that it matches the data type of the returned value.

The last two items to note concern the coding of the find_max( ) function. The header line of find_max( ) defines that the function will return a floating point value and declares the order, names, and data types of the arguments required by the function. Finally, the expression in the return statement evaluates to the correct return value data type defined in the function's header line. Thus, find_max( ) is internally consistent in sending a floating point value back to any function that might be used to call it, and from the calling side main( ) has been correctly alerted to receive and use the returned value.

In writing your own functions you must always keep these four items in mind. See if you can identify these four items in Program 7-3.

Program 7-3

```
#include <stdio.h>
main( )
{
  int count;                    /* start of declarations */
  double fahren;
  double tempvert(double);      /* function prototype    */

  for(count = 1; count <= 4; ++count)
  {
    printf("Enter a Fahrenheit temperature: ");
    scanf("%lf", &fahren);
    printf("The Celsius equivalent is %6.2f\n\n", tempvert(fahren) );
  }
}

double tempvert(double in_temp)    /* function header */
{
  return( (5.0/9.0) * (in_temp - 32.0) );
}
```

In reviewing Program 7-3 let us first analyze the function tempvert( ). The complete definition of the function begins with the function's header line and ends with the closing brace after the return statement. The function is declared as a double, which means the expression in the function's return statement must evaluate to a double precision number. Since a function header line is not a statement but the start of the code defining the function, the header line does not end with a semicolon.

Within `tempvert( )`, `in_temp` is declared as a double precision parameter. Since an expression with a double precision number yields a double precision value, the correct data type, a double precision number, is returned by the function.

On the receiving side, `main( )` has a function prototype statement for the function `tempvert( )` that agrees with `tempvert( )`'s function definition. As with all declaration statements, which is what a function prototype is, multiple declarations of the same type may be made within the same statement. Thus, we could have used the same declaration statement for `tempvert( )`'s function prototype and the variable `fahren`. If we had done so, the single declaration statement

```
double fahren, tempvert(double);
```

could have been used to replace the two individual declarations for `tempvert( )` and `fahren`. For clarity, however, we will always keep function prototype statements apart from variable declaration statements. No additional variable is declared in `main( )` to store the returned value from `tempvert( )` because the returned value is immediately passed to `printf( )` for display.

Placing user-written function after the `main( )` function, as is done in both Program 7-2 and Program 7-3, is a matter of choice. Some programmers prefer to put all called functions at the top of a program and make `main( )` the last function listed. We prefer to list `main( )` first because it is the driver function that should give anyone reading the program an idea of what the complete program is about before encountering the details of each function. Either placement approach is acceptable and you will encounter both styles in your programming work. In no case, however, can a user-written function be placed inside another function. Each C function is a separate and independent entity with its own parameters and variables and must never be included within another function.

Standard Library Functions

In addition to writing their own function, all C programmers have access to a standard, preprogrammed set of functions for handling input and output of data, computing mathematical quantities, and manipulating strings of characters. These preprogrammed functions are stored in a system library that contains the collection of standard and tested functions available on your system.

Before using these system-defined functions, you must know:

the name of each available function
the arguments required by each function
the data type of the result (if any) returned by each function
a description of what each function does

The first three items are provided by the function header. For example, consider the function named `sqrt( )`, which calculates the square root of its argument. The function header for this function is:

```
double sqrt(double num)
```

This header lists all the information required to call the `sqrt( )` function. `sqrt( )` expects a double precision argument and returns a double precision value.

Many library functions require a standard set of common declarations and other information for proper operation. This information is always contained in a standard header file. To include the information in the appropriate header file in your program, you must include the following statement in your program before the desired library function is called:

```
#include <header-file-name>      ← no semicolon
```

If you intend to use a library function in your program, placing the appropriate `#include` statement at the top of the program ensures proper access to the library function to all subsequent functions in your program. As we saw in Chapter 3, using the mathematical library functions requires the statement

```
#include <math.h>
```

Input/Output Library Functions

We have already made extensive use of two input/output (I/O) library functions, `printf( )` and `scanf( )`. In this section we present two additional I/O library routines. These two routines are written and contained within the header file `stdio.h`; thus they require inclusion of the `stdio.h` header file in any program that uses them.[4]

The `getchar( )` routine can be used for single character input. The function header for `getchar( )` is:

```
int getchar(void)
```

`getchar( )` expects no arguments to be passed to it and returns an integer data type. Returning characters in integer format allows the end-of-file (`EOF`) Sentinel previously described in Section 5.2 to be returned. The `EOF` sentinel has an integer code. If this sentinel is to be correctly recognized as input, `getchar( )` must return integer values. The `getchar( )` routine is used to return the next single character entered at the terminal. For example, a statement such as

```
in_char = getchar( );
```

causes the next character entered at the terminal to be stored in the variable `in_char`. This is equivalent to the longer statement `scanf("%d", &in_char);`. The `getchar( )` routine is extremely useful when continuously inputting strings of characters, which is the topic of Chapter 11.

The output library function corresponding to `getchar( )` is the `putchar( )` function. The `putchar( )` function expects a single character argument and displays the character passed to it on the terminal. For example, the statement `putchar('a')` causes the letter `a` to be displayed on the standard output device—it is equivalent to the longer statement `printf("%c",'a')`.

String Library Functions

Almost all C compilers have an extensive set of library functions for the input, comparison, manipulation, and output of strings of characters. A list and

[4] Formally, these two routines are created as *macros* within the `stdio.h` header file. The writing of macros is presented in Section 14.3.

TABLE 7–1 String Library Functions

Name	Description
`strcat(string1,string2)`	Concatenate `string2` to `string1`.
`strchr(string,character)`	Locate the position of the first occurrence of the character within the string.
`strcmp(string1,string2)`	Compare `string2` to `string1`.
`strcpy(string1,string2)`	Make `string1` equal to `string2`.
`strlen(string)`	Determine the length of the string.

description of these functions is given in Table 7–1. We will describe these functions in Chapter 11, where character strings are presented in detail. Use of these functions requires inclusion of the `string.h` header file.

Miscellaneous Routines

In addition to the input/output, mathematical, and string functions, all system libraries have an extensive collection of miscellaneous functions and other routines. Some of the more useful of these are listed in Table 7–2.

The header lines for these routines are included in a standard file named `ctype.h`. To access and use them in a program may require the following statement before `main( )`:

```
#include <ctype.h>
```

The routines listed in Table 7–2 are particularly useful in checking characters input by a user. For example, Program 7-4 continuously requests that a user enter

TABLE 7–2 Miscellaneous Routines

Name	Description
`isalpha(character)`	Returns a nonzero number if the character is a letter; otherwise it returns a zero.
`isupper(character)`	Returns a nonzero number if the character is uppercase; otherwise it returns a zero.
`islower(character)`	Returns a nonzero number if the character is lowercase; otherwise it returns a zero.
`isdigit(character)`	Returns a nonzero number if the character is a digit (0 through 9); otherwise it returns a zero.
`toupper(character)`	Returns the uppercase equivalent if the character is lowercase; otherwise it returns the character unchanged.
`tolower(character)`	Returns the lowercase equivalent if the character is uppercase; otherwise it returns the character unchanged.

a character and determines if the character is a letter or a digit. The program exits the `while` loop when an `f` is typed. So that the user won't have to decide whether a lowercase or uppercase `f` must be entered to stop the program, the program converts all input to lowercase and just checks for a lowercase `f`.

Program 7-4

```
#include <stdio.h>
#include <ctype.h>
main( )
{
  char in_char;

  do
  {
    printf("\nPush any key (type an f to stop) ");
    in_char = getchar( );      /* get the next character typed */
    in_char = tolower(in_char);       /* convert to lowercase */
    getchar( );                /* get and ignore the ENTER key */
    if ( isalpha(in_char) )    /* a nonzero value is true in C */
      printf("\nThe character entered is a letter.");
    else if ( isdigit(in_char) )
      printf("\nThe character entered is a digit.");
  } while (in_char != 'f');

}
```

A few remarks are in order in reviewing Program 7-4. First, the condition being tested in the `if-else` statement makes use of the fact that a condition is considered true if it evaluates to a nonzero value. Thus, the condition `( isalpha(in_char) )` could have been written as `( isalpha(in_char) != 0 )`, and the condition `( isdigit(in_char) )` could have been written `(isdigit(in_char) != 0 )`. The second call to `getchar( )` in the `do-while` loop is used to remove the ENTER key.

Since functions return values, a function may itself be an argument to a function (including itself). For example, these two statements in Program 7-4,

```
in_char = getchar( );      /* get the next character typed */
in_char = tolower(in_char);       /* convert to lowercase */
```

may be combined into the single statement:

```
in_char = tolower(getchar( ));
```

Exercises 7.1

1. For the following function headers, determine the number, type, and order (sequence) of values that should be passed to the function when it is called and the data type of the value returned by the function:
   a. `int factorial(int n)`
   b. `double price(int type, double yield, double maturity)`
   c. `double yield(int type, double price, double maturity)`
   d. `char interest(char flag, float price, float time)`
   e. `float total(float amount, float rate)`
   f. `float roi(int a, int b, char c, char d, float e, float f)`
   g. `void get_val(int item, int iter, char decflag)`

2. Write function headers for the following:
   a. a function named `check( )`, which has three arguments. The first argument should accept an integer number, the second argument a floating point number, and the third argument a double precision number. The function returns no value.
   b. a function named `find_abs( )` that accepts a double-precision number passed to it and displays its absolute value.
   c. a function named `mult( )` that accepts two floating point numbers as arguments, multiplies these two numbers, and returns the result.
   d. a function named `sqr_it( )` that computes and returns the square of the integer value passed to it.
   e. a function named `powfun( )` that raises an integer number passed to it to a positive integer power and returns the result.
   f. a function that produces a table of the numbers from 1 to 10, their squares, and their cubes. No arguments are to be passed to the function and the function returns no value.

3. Write C function prototypes corresponding to each of the function header lines given in Exercise 1.

4a. Write a C function named `check( )`, which has three arguments. The first argument should accept an integer number, the second argument a floating point number, and the third argument a double precision number. The body of the function should just display the values of the data passed to the function when it is called.

(**Note:** ***When tracing errors in functions, it is very helpful to have the function display the values it has been passed. Frequently the error is not in what the body of the function does with the data, but in the data received and stored. This type of error occurs when a different data type is passed to the function from the data type declared for the arguments.***)

b. Include the function written in Exercise 4a in a working program. Make sure your function is called from `main( )`. Test the function by passing various data to it that are of a different data type then that specified for each parameter.

5a. Write a C function named `find_abs( )` that accepts a double precision number passed to it, computes its absolute value, and returns the absolute value to the calling function.
b. Include the C function written in Exercise 5a in a working program. Make sure your function is called from `main( )` and correctly returns a value to `main( )`. Have `main( )` use a function prototype to declare the `find_abs( )` function and use `printf( )` to display the value returned. Test the function by passing various data to it.

6a. Write a C function called `mult( )` that accepts two double precision numbers as parameters, multiplies these two numbers, and returns the result to the calling function.
b. Include the function written in Exercise 6a in a working program. Make sure your function is called from `main( )` and correctly returns a value to `main( )`. Have `main( )`

use a function prototype to declare the `mult( )` function and use `printf( )` to display the value returned. Test the function by passing various data to it.

7a. Write a C function named `sqr_it( )` that computes the square of the value passed to it and displays the result. The function should be capable of squaring numbers with decimal points.
b. Include the function written in Exercise 7a in a working program. Make sure your function is called from `main( )`. Test the function by passing various data to it.

8a. Write a C function named `powfun( )` that raises an integer number passed to it to a positive integer power and displays the result. The positive integer should be the second value passed to the function. Declare the variable used to store the result as a long integer data type to ensure sufficient storage for the result.
b. Include the function written in Exercise 8a in a working program. Make sure your function is called from `main( )`. Test the function by passing various data to it.

9a. Write a C function that produces a table of the numbers from 1 to 10, their squares, and their cubes. The function should produce the same display as that produced by Program 5-12.
b. Include the function written in Exercise 9a in a working program. Make sure your function is called from `main( )`. Test the function by passing various data to it.

10a. Modify the function written for Exercise 9 to accept the starting value of the table, the number of values to be displayed, and the increment between values. Name your function `sel_tab( )`. A call to `sel_tab(6,5,2);` should produce a table of 6 lines, the first line starting with the number 5 and each succeeding number increasing by 2.
b. Include the function written in Exercise 10a in a working program. Make sure your function is called from `main( )`. Test the function by passing various data to it.

11a. Rewrite the function `tempvert( )` in Program 7-3 to accept a temperature and a character as arguments. If the character passed to the function is the letter `f`, the function should convert the passed temperature from Fahrenheit to Celsius, else the function should convert the passed temperature from Celsius to Fahrenheit.
b. Modify the `main( )` function in Program 7-3 to call the function written for Exercise 11a. Your `main( )` function should ask the user for the type of temperature being entered and pass the type (`f` or `c`) into `tempvert( )`.

12a. A second-degree polynomial in x is given by the expression $ax^2 + bx + c$, where a, b, and c are known numbers and a is not equal to zero. Write a function named `poly_two(a,b,c,x)` that computes and returns the value of a second-degree polynomial for any passed values of `a`, `b`, `c`, and `x`.
b. Include the function written in Exercise 12a in a working program. Make sure your function is called from `main( )` and correctly returns a value to `main( )`. Have `main( )` use a function prototype to declare the `poly_two( )` function and use `printf( )` to display the value returned. Test the function by passing various data to it.

13a. An extremely useful programming algorithm for rounding a real number to n decimal places is:

Step 1. Multiply the number by 10^n
Step 2. Add .5
Step 3. Delete the fractional part of the result
Step 4. Divide by 10^n

For example, using this algorithm to round the number 78.374625 to three decimal places yields:

Step 1: $78.374625 \times 10^3 = 78374.625$
Step 2: $78374.625 + .5 = 78375.125$
Step 3: Deleting the fractional part = 78375
Step 4: 78375 divided by $10^3 = 78.375$

Using this algorithm, write a program that accepts a user-entered value of money, multiplies the entered amount by an 8.675% interest rate, and displays the result rounded to two decimal places.

b. Enter, compile, and execute the program written for Exercise 13a.

14a. Write a function named `whole( )` that returns the integer part of any number passed to the function.

b. Include the function written in Exercise 14a in a working program. Make sure your function is called from `main( )` and correctly returns a value to `main( )`. Have `main( )` use `printf( )` to display the value returned. Test the function by passing various data to it.

15a. Write a C function named `fracpart( )` that returns the fractional part of any number passed to the function. For example, if the number `256.879` is passed to `fracpart( )`, the number `.879` should be returned. Have the function `fracpart( )` call the function `whole( )` that you wrote in Exercise 14. The number returned can then be determined as the number passed to `fracpart( )` less the returned value when the same argument is passed to `whole( )`. The completed program should consist of `main( )` followed by `fracpart( )` followed by `whole( )`.

b. Include the function written in Exercise 15a in a working program. Make sure your function is called from `main( )` and correctly returns a value to `main( )`. Have `main( )` use `printf( )` to display the value returned. Test the function by passing various data to it.

7.2 Arrays as Arguments

Individual array elements are passed to a called function in the same manner as individual scalar variables; they are simply included as subscripted variables when the function call is made. For example, the function call `find_min (grades[2], grades[6]);` passes the values of the elements `grades[2]` and `grades[6]` to the function `find_min( )`.

Passing a complete array of values to a function is in many respects an easier operation than passing individual elements. The called function receives access to the actual array, rather than a copy of the values in the array. For example, if grades is an array, the function call `find_max(grades);` makes the complete grades array available to the `find_max( )` function. This is different from passing a single variable to a function.

Recall that when a single scalar argument is passed to a function, the called function only receives a copy of the passed value, which is stored in one of the function's parameters. If arrays were passed in this manner, a copy of the complete array would have to be created. For large arrays, making duplicate copies of the array for each function call would waste computer storage space and would frustrate the effort to return multiple element changes made by the called program (recall that a function returns at most one value). To avoid these problems, the called function is given direct access to the original array. Thus, any changes made by the called function are made directly to the array itself. For the following specific examples of function calls, assume that the arrays `nums`, `keys`, `units`, and `prices` are declared as:

```
int nums[5];                          /* an array of five integers   */
char keys[256];                       /* an array of 256 characters */
double units[500], prices[500];       /* two arrays of 500 doubles   */
```

For these arrays, the following function calls can be made:

```
find_max(nums);
find_ch(keys);
calc_tot(nums, units, prices);
```

In each case, the called function receives direct access to the named array.

On the receiving side, the called function must be alerted that an array is being made available. For example, suitable function header lines for the previous functions are:

```
find_max(int vals[5])
find_ch(char in_keys[256])
calc_tot(int arr1[5], double arr2[500], double arr3[500])
```

In each of these function header lines, the names in the parameter list are chosen by the programmer. However, the parameter names used by the functions still refer to the original array created outside the function. This is made clear in Program 7-5.

Program 7-5

```
#include <stdio.h>
main( )
{
  int nums[5] = {2, 18, 1, 27, 16};
  int find_max(int [5]);        /* function prototype */

  printf("The maximum value is %d", find_max(nums));
}

int find_max(int vals[5])          /* find the maximum value */
{
  int i, max = vals[0];

  for (i = 1; i <= 4; ++i)
    if (max < vals[i]) max = vals[i];

return (max);
}
```

Notice that the function prototype for `find_max( )` within `main( )` declares that `find_max` will return an integer and expects an array of five integers as an actual argument. It is also important to know that only one array is created in Program 7-5. In `main( )` this array is known as `nums`, and in `find_max( )` the array is known as `vals`. As illustrated in Figure 7–8, both names refer to the same array. Thus, in Figure 7–8 `vals[3]` is the same element as `nums[3]`.

The parameter declaration in the `find_max( )` header line actually contains

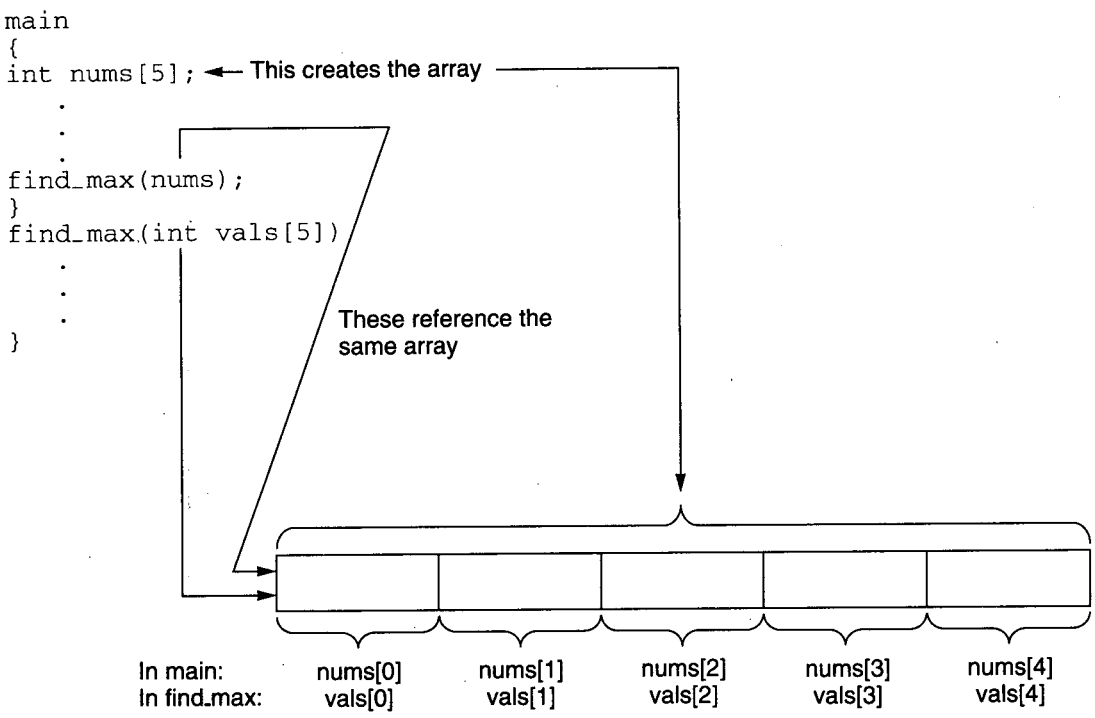


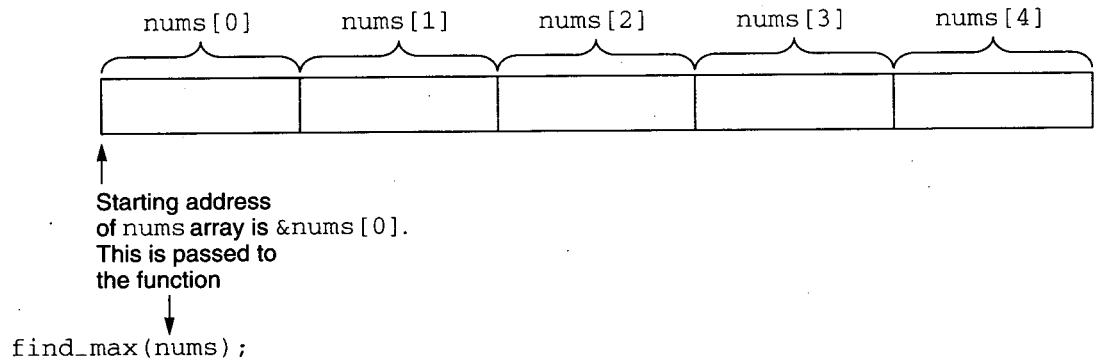
FIGURE 7–8 Only One Array Is Created

extra information that is not required by the function. All that `find_max( )` must know is that the argument `vals` references an array of integers. Since the array has been created in `main( )` and no additional storage space is needed in `find_max( )`, the declaration for `vals` can omit the size of the array. Thus, an alternative function header line is:

```
find_max(int vals[ ])
```

This form of the function header makes more sense when you realize that only one item is actually passed to `find_max( )` when the function is called: the starting address of the `num` array. This is illustrated in Figure 7–9.

FIGURE 7–9 The Starting Address of the Array Is Passed



Since only the starting address of `vals` is passed to `find_max( )`, the number of elements in the array need not be included in the declaration for vals.[5] In fact, it is generally advisable to omit the size of the array in the function header line. For example, consider the more general form of `find_max( )`, which can be used to find the maximum value of an integer array of arbitrary size:

```
int find_max(int vals[ ], int num_els) /* find the maximum value */
{
  int i, max = vals[0];

  for (i = 1; i < num_els; ++i)
   if (max < vals[i]) max = vals[i];

  return(max);
}
```

The more general form of `find_max( )` declares that the function returns an integer value. The function expects the starting address of an integer array and the number of elements in the array as arguments. Then, using the number of elements as the boundary for its search, the function's `for` loop causes each array element to be examined in sequential order to locate the maximum value. Program 7-6 illustrates the use of `find_max( )` in a complete program.[6]

The output displayed when Program 7-6 is executed is:

```
The maximum value is 27
```

Passing two-dimensional arrays into functions is a process identical to passing single-dimensional arrays. Assuming that the following two-dimensional arrays named `test`, `code`, and `stocks` are declared as:

```
int test[7][9];
char code[26][10];
float stocks[256][52];
```

the following function calls are valid:

```
find_max(test);
obtain(code);
price(stocks);
```

On the receiving side, the called function must be alerted that a two-dimensional array is being made available. For example, suitable function header lines for the previous functions are:

---

[5] An important consequence of this is that `find_max( )` has direct access to the passed array. This means that any change to an element of the `vals` array actually is a change to the `nums` array. This is significantly different from the situation with scalar variables, where the called function does not receive direct access to the passed variable.

[6] If you are using a non-ANSI C compiler, the term `static` must be placed before the word `int` in the declaration statement for `nums`.

Program 7-6

```
#include <stdio.h>
main( )
{
  int nums[5] = {2, 18, 1, 27, 16};
  int find_max(int [ ], int);   /* function prototype */

  printf("The maximum value is %d", find_max(nums,5));
}

int find_max(int vals[ ], int num_els)
{
  int i, max = vals[0];

  for (i = 1; i < num_els; ++i)
    if (max < vals[i]) max = vals[i];

  return(max);
}
```

```
find_max(int nums[7][9]
obtain(char key[26][10])
price(float names[256][52])
```

In each of these function header lines, the argument names chosen are local to the function. However, the internal local names used by the function still refer to the original array created outside the function. Program 7-7 illustrates passing a local, two-dimensional array into a function that displays the array's values.

Only one array is created in Program 7-7. This array is known as `val` in `main( )` and as `nums` in `display( )`. Thus, `val[0][2]` refers to the same element as `nums[0][2]`.

Program 7-7

```
#include <stdio.h>
main( )
{
  int val[3][4] = {8,16,9,52,
                   3,15,27,6,
                  14,25,2,10};
  void display(int [3][4]);    /* function prototype */
  display(val);
}

void display(int nums[3][4])
```

Continued

```
{
  int row_num, col_num;

  for (row_num = 0; row_num < 3; ++row_num)
  {
    for(col_num = 0; col_num < 4; ++col_num)
      printf("%4d",nums[row_num][col_num]);
    printf("\n");
  }
}
```

Notice the use of the nested `for` loop in Program 7-7. Nested `for` statements are especially useful when dealing with multidimensional arrays because they allow the programmer to cycle through each element. In Program 7-7, the variable `row_num` controls the outer loop and the variable `col_num` controls the inner loop. For each pass through the outer loop, which corresponds to `row`, the inner loop makes one pass through the column elements. After a complete column is printed, the `\n` escape sequence causes a new line to be started for the next row. The effect is a display of the array in a row-by-row fashion:

```
  8  16   9  52
  3  15  27   6
 14  25   2  10
```

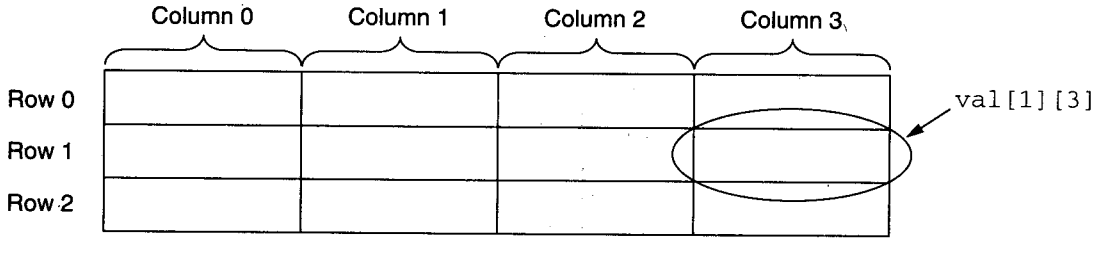
The argument declaration for `nums` in `display( )` contains extra information that is not required by the function. The declaration for `nums` can omit the row size of the array. Thus, an alternative function declaration is:

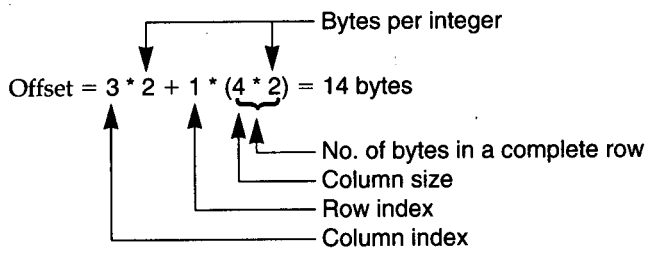
```
display(int nums[][4])
```

The reason why the column size must be included while the row size is optional becomes obvious when you consider how the array elements are stored in memory. Starting with element `val[0][0]`, each succeeding element is stored consecutively, row by row, as `val[0][0]`, `val[0][1]`, `val[0][2]`, `val[0][3]`, `val[1][0]`, `val[1][1]`, etc., as illustrated in Figure 7–10.

As with all array accesses, an individual element of the `val` array is obtained by adding an offset to the starting location of the array. For example, the element `val[1][3]` is located at an offset of 14 bytes from the start of the array. Internally, the computer uses the row index, column index, and column size to determine this offset using the following calculation:

FIGURE 7–10 Storage of the `val` Array





The number of columns is necessary in the offset calculation so that the computer can determine the number of positions to skip over in order to get to the desired row.

Exercises 7.2

1. The following declaration was used to create the `prices` array:

```
double prices[500];
```

Write two different function header lines for a function named `sort_arr( )` that accepts the `prices` array as an argument named `in_array` and returns a double value.

2. The following declaration was used to create the `keys` array:

```
char keys[256];
```

Write two different function header lines for a function named `find_key( )` that accepts the `keys` array as an argument named `select` and returns no value.

3. The following declaration was used to create the `rates` array:

```
float rates[256];
```

Write two different function header lines for a function named `prime( )` that accepts the `rates` array as an argument named `rates` and returns a floating point value.

4a. Modify the `find_max( )` function in Program 7-5 to locate the minimum value of the passed array and returns a floating point value.

b. Include the function written in Exercise 4a in a complete program and run the program on a computer.

5. Write a C program that has a declaration in `main( )` to store the following numbers into an array named `rates`: 6.5, 7.2, 7.5, 8.3, 8.6, 9.4, 9.6, 9.8, 10.0. There should be a function call to `show( )` that accepts the `rates` array as an argument named `rates` and then displays the numbers in the array.

6a. Write a C program that has a declaration in `main( )` to store the string `"Vacation is near"` into an array named `message`. There should be a function call to `display( )` that accepts `message` in an argument named `strng` and then displays the message.

b. Modify the `display( )` function written in Exercise 6a to display the first eight elements of the `message` array.

7. Write a C program that declares three single-dimensional arrays named `price`, `quantity`, and `amount`. Each array should be declared in `main( )` and should be capable of holding ten double precision numbers. The numbers that should be stored in `price` are 10.62, 14.89, 13.21, 16.55, 18.62, 9.47, 6.58, 18.32, 12.15, 3.98. The numbers that should be stored in `quantity` are 4, 8.5, 6, 7.35, 9, 15.3, 3, 5.4, 2.9, 4.8. Your program should pass

these three arrays to a function called `extend( )`, which should calculate the elements in the `amount` array as the product of the corresponding elements in the `price` and `quantity` arrays (for example, `amount[1] = price[1] * quantity[1]`). After `extend( )` has put values into the `amount` array, the values in the array should be displayed from within `main( )`.

8. Write a C program that includes two functions named `calc_avg( )` and `variance( )`. The `calc_avg( )` function should calculate and return the average of the values stored in an array named `testvals`. The array should be declared in `main( )` and include the values 89, 95, 72, 83, 99, 54, 86, 75, 92, 73, 79, 75, 82, 73. The `variance( )` function should calculate and return the variance of the data. The variance is obtained by subtracting the average from each value in `testvals`, squaring the values obtained, adding them, and dividing by the number of elements in `testvals`. The values returned from `calc_avg( )` and `variance` should be displayed using `printf( )` function calls in `main( )`.

9a. Write a function that finds and displays the maximum value in a two-dimensional array of integers. The array should be declared as a 10 row by 20 column array of integers in `main( )` and the starting address of the array should be passed to the function.

b. Modify the function written in Exercise 9a so that it also displays the row and column numbers of the element with the maximum value.

c. Can the function you wrote for Exercise 9a be generalized to handle any size two-dimensional array?

10. Write a function that can be used to sort the elements of a 10 by 20 two-dimensional array of integers. (*Hint:* Review Section 6.7 before doing this exercise.)

7.3 Applications

In this section we present two applications using functions. In the first application we create a random number generator function and incorporate it in a C program that produces a series of 10 random numbers. In the second application the random number generator is used to simulate a coin tossing experiment.

Application 1: Random Number Generation

There are many mathematical and engineering problems in which probability must be considered or statistical sampling techniques must be used. For example, in simulating automobile traffic flow or telephone usage patterns, statistical models are required. Additionally, applications such as simple computer games and more involved "strategy games" in business and science can only be described statistically. All of these statistical models require the generation of random numbers.

One method of generating random numbers is the power residue method. In one version of this method a suitable n digit "seed" number, where n is an even number, is multiplied by the value $(10^{n/2} - 3)$. Using the lowest n digits of the result (the "residue") produces a new seed. Continuing this procedure produces a series of random numbers, with each new number used as the seed for the next number. If the original seed has four or more digits (n equal to or greater than 4) and is not divisible by either 2 or 5, this procedure yields $5 \times 10^{(n-2)}$ random numbers before a sequence of numbers repeats itself. For example, starting with a 6-digit seed ($n = 6$) such as 654321, a series of $5 \times 10^4 = 50{,}000$ random numbers can be generated. Using a power residue algorithm we will write a random number generator function.

The specific power residue algorithm employed consists of the following steps:

Step 1: Have the user enter a 6-digit integer seed that is not divisible by 2 or 5—this means the number should be an odd number not ending in 5.
Step 2: Multiply the seed number by 997, which is $10^3 - 3$.
Step 3: Extract the lower 6 digits of the result produced by Step 2. Use this random number as the next seed.
Step 4: Repeat Steps 2 and 3 for as many random numbers as needed.

Thus, if the user-entered seed number is 654321 (Step 1), the first random number generated is calculated as follows:

Step 2: 654321 * 997 = 652358037
Step 3: Extract the lower 6 digits of the number obtained in Step 2. This is accomplished using a standard programming "trick." The trick involves:
  Step 3a. Dividing the number by 10^6 = 1000000
    For example, 652358037 / 1000000 = 652.358037
  Step 3b. Taking the integer part of the result of Step 3a.
    For example, the integer part of 652.358037 = 652
  Step 3c. Multiplying the previous result by 10^6
    For example, $652 \times 10^6 = 652000000$
  Step 3d. Subtracting this result from the original number
    For example, 652358037 − 652000000 = 358037

The integer part of a floating point number can be taken either by assigning the floating point number to an integer variable, or by a C *cast* (see Section 3.1). For now, we will simply assign the floating point number to an integer variable. Thus, the algorithm for producing a random number can be accomplished using the following code:

```
/* make sure i is declared as an integer variable */
i = 997.0 * x / 1.e6;  /* take the integer part */
x = 997.0 * x - i * 1.e6; /* extract the last 6 digits*/
```

Program 7-8 (p. 300) uses this algorithm in a random number generator function to produce a series of 10 random numbers.

Following is a sample run using Program 7-8.

```
Enter an odd 6 digit number not ending in 5: 654321

358037.000000
962889.000000
   333.000000
332001.000000
  4997.000000
982009.000000
 62973.000000
784081.000000
728757.000000
570729.000000
```

Program 7-8

```
#include <stdio.h>
main( )
/* this program generates ten pseudo-random numbers from */
/* a user input "seed" value                              */
{
  int i;
  float seed;
  float rand(float);

  printf("\nEnter an odd 6 digit number not ending in 5: ");
  scanf("%f", &seed);
  for (i = 1; i <= 10; ++i)
  {
    seed = rand(seed);
    printf("\n%14.6f",seed);
  }
}
float rand(float x)
{
  int i;

  i = 997.0 * x / 1.e6;      /* take the integer part     */
  x = 997.0 * x - i * 1.e6; /* extract the last 6 digits*/
  return(x);
}
```

Conventionally, random number generators are used to produce random numbers within the range 0.0 to 1.0. To produce such numbers using Program 7-8's `rand( )` function simply requires dividing the returned value by 10^6.

Application 2: Coin Toss Simulation

A common use of random numbers is to simulate events using a program, rather than going through the time and expense of constructing a real-life experiment. For example, statistical theory tells us that the probability of having a tossed coin turn up heads is 1/2. Similarly, there is a 50 percent probability of having a single tossed coin turn up tails.

Using these probabilities we would expect a single coin that is tossed 1000 times to turn up heads 500 times and tails 500 times. In practice, however, this is never exactly realized for a single experiment consisting of 1000 tosses. Instead of actually tossing a coin 1000 times, however, we can use a random number generator to simulate these tosses. In particular, we will use the random number function developed in the previous application.

Since the function `rand( )` function created in the previous application returns a random number between 0 and 999999, we divide the returned value by 1000000 to produce a random number between 0 and 1. Using this "normalized" random number, the algorithm to simulate 1000 coin tosses is given by this pseudocode:

```
Initialize the heads count to zero
Initialize the tails count to zero
For 1000 times
   get a random number between 0 and 1
   if the random number is greater than .5
      consider this as a head and
      add one to the heads count
   else
      consider this as a tail and
      add one to the tails count
   endif
Endfor
Calculate the percentage of heads as
   the number of heads divided by 1000 x 100%
Calculate the percentage of tails as
   the number of tails divided by 1000 x 100%
Print the percentage of heads and tails obtained
```

Program 7-9 codes this algorithm in C.

Following are two sample runs using Program 7-9 (p. 302):

```
Enter an odd 6 digit number not ending in 5: 654321

Heads came up 49.599998 percent of the time
Tails came up 50.400002 percent of the time
```

and

```
Enter an odd 6 digit number not ending in 5: 876543

Heads came up 51.299999 percent of the time
Tails came up 48.700001 percent of the time
```

Writing and executing Program 7-9 is certainly easier than manually tossing a coin 1000 times. It should be noted that the validity of the results produced by the program depends on how random the numbers produced by the random number function actually are.

Program 7-9

```
#include <stdio.h>
main( )
/* a program to simulate the tossing of a coin 1000 times */
{
  int heads, tails, i;
  float seed, x, flip, perheads, pertails;
  float rand(float);       /* function prototype */

  printf("\nEnter an odd 6 digit number not ending in 5: ");
  scanf("%f", &seed);
  heads = 0;       /* initialize heads count */
  tails = 0;       /* initialize tails count */
  /* simulate 1000 tosses of a coin */
  for (i = 1; i <= 1000; ++i)
  {
    seed = rand(seed);
    flip = seed / 1.e6;  /* normalize the number between 0 and 1 */
    if (flip > 0.5)
      heads = heads + 1;
    else
      tails = tails + 1;
  }
  perheads = (heads / 1000.0) * 100.0; /* calculate heads percentage */
  pertails = (tails / 1000.0) * 100.0; /* calculate tails percentage */
  printf("\nHeads came up %f percent of the time", perheads);
  printf("\nTails came up %f percent of the time", pertails);
}
float rand(float x)
{
  int i;
  i = 997.0 * x / 1.e6;     /* take the integer part */
  x = 997.0 * x - i * 1.e6;
  return(x);
}
```

Exercises 7.3

1. Modify Program 7-9 so that it requests the number of tosses from the user. (*Hint:* Make sure to have the program correctly determine the percentages of heads and tails obtained.)

2. (Central Limit Theorem Simulation) Modify Program 7-9 so that it automatically generates 20 simulations, with each simulation having 1000 tosses. Print out the percentage for each run and the percentages for the 20 runs combined.

3a. Write a C program that uses a user-entered 6-digit integer to produce a random number between 1 and 100. The program should then give the user 7 tries to guess the

generated random number. If the user guesses the correct number the message `"HOORAY! YOU WIN!"` should be displayed. After each incorrect guess the computer should display the message `"WRONG NUMBER - TRY AGAIN"` and indicate the number of guesses left. After 7 incorrect guesses the computer should display the message `"SORRY YOU LOSE"`. (*Hint:* To generate a number between 1 and 100 from a random number function `rand(x)` that generates numbers between 0.0 and 1.0 requires the use of the expression `1 + (int)(100 * rand(x))`.

b. Modify the program written for Exercise 3a to allow the user to run the game again after a game has been completed. The program should display the message `"WOULD YOU LIKE TO PLAY AGAIN 'Y'/'N'?: "` and restart if the user enters either `'Y'` or `'y'`.

4. In the game of blackjack the cards 2 through 10 are counted at their face values, regardless of suit; all picture cards (jack, queen, and king) are counted as 10; and an ace is counted as either 1 or 11, depending on the total count of all the cards in a player's hand. The ace is counted as 11 only if the total value of all cards in a player's hand does not exceed 21, else it is counted as 1. Using this information write a C program that uses a random number generator to select three cards (a 1 initially corresponding to an ace, a 2 corresponding to a face card of 2, and so on), calculate the total value of the hand appropriately, and display the value of the three cards with a printed message.

5. Write a C function that determines the quadrant that a line drawn from the origin resides in. The determination of the quadrant is made using the angle that the line makes with the positive *X* as follows:

Angle from the Positive *X* Axis	Quadrant
Between 0 and 90 degrees	1
Between 90 and 180 degrees	2
Between 180 and 270 degrees	3
Between 270 and 360 degrees	4

Note: ***If the angle is exactly 0, 90, 180, or 270 degrees the corresponding line does not reside in any quadrant but lies on an axis. For this case your function should return a zero.***

6. All years that are evenly divisible by 400 or are evenly divisible by 4 and not evenly divisible by 100 are leap years. For example, since 1600 is evenly divisible by 400, the year 1600 was a leap year. Similarly, since 1988 is evenly divisible by 4 but not by 100, the year 1988 was also a leap year. Using this information, write a C function that accepts the year as a user input and returns a 1 if the passed year is a leap year or a 0 if it is not.

7. Based on an automobile's model year and weight the state of New Jersey determines the car's weight class and registration fee using the following schedule:

Model Year	Weight	Registration Fee
1970 or earlier	less than 2,700 lbs	$16.50
	2,700 to 3,800 lbs	25.50
	more than 3,800 lbs	46.50
1971 to 1979	less than 2,700 lbs	27.00
	2,700 to 3,800 lbs	30.50
	more than 3,800 lbs	52.50
1980 or later	less than 3,500 lbs	19.50
	3,500 or more lbs	52.50

Using this information, write a C subprogram function that accepts the year and weight of an automobile and returns the registration fee for the car.

7.4 Variable Scope

Now that we have begun to write programs containing more than one function, we can look more closely at the variables declared within each function and their relationship to variables in other functions.

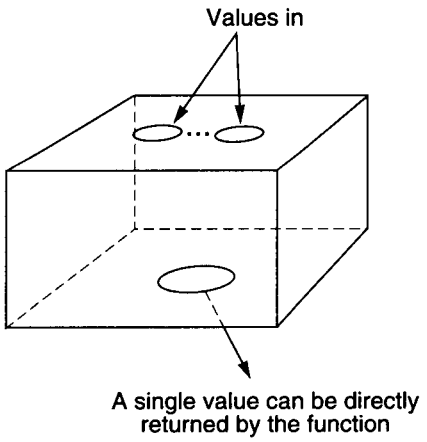
By their very nature, C functions are constructed to be independent modules. As we have seen, values are passed to a function using the function's argument list and a value is returned from a function using a `return` statement. Seen in this light, a function can be thought of as a closed box with slots at the top to receive values and a single slot at the bottom of the box to directly return a value (see Figure 7–11).

The metaphor of a closed box is useful because it emphasizes the fact that what goes on inside the function, including all variable declarations within the function's body, are hidden from the view of all other functions. Since the variables created inside a function are available only to the function itself, they are said to be local to the function, or local variables. This term refers to the scope of a variable, where scope is defined as the section of the program where the variable is valid or "known." A variable can have either a local scope or a global scope. A variable with a local scope is simply one that has had storage locations set aside for it by a declaration statement made within a function body. Local variables are only meaningful when used in expressions or statements inside the function that declared them. This means that the same variable name can be declared and used in more than one function. For each function that declares the variable, a separate and distinct variable is created.

All the variables we have used until now have been local variables. This is a direct result of placing our declaration statements inside functions and using them as definition statements that cause the computer to reserve storage for the declared variable. As we shall see, declaration statements can be placed outside functions and need not act as definitions that cause new storage areas to be reserved for the declared variable.

A variable with global scope, more commonly termed a global variable, is one

FIGURE 7–11 A Function Can Be Considered a Closed Box



whose storage has been created for it by a declaration statement located outside any function. Global variables are also referred to as external variables. These variables can be used by all functions in a program that are physically placed after the global variable declaration. This is shown in Program 7-10, where we have purposely used the same variable name inside both functions contained in the program.

The variable `firstnum` in Program 7-10 is a global variable because its storage is created by a declaration statement located outside a function. Since both functions, `main( )` and `valfun( )`, follow the declaration of `firstnum`, both of these functions can use this global variable with no further declaration needed.

Program 7-10 also contains two separate local variables, both named `secnum( )`. Storage for the `secnum` variable named in `main( )` is created by the declaration statement located in `main( )`. A different storage area for the `secnum` variable in `valfun( )` is created by the declaration statement located in the `valfun( )` function. Figure 7–12 illustrates the three distinct storage areas reserved by the three declaration statements found in Program 7-10.

Each of the variables named `secnum` are local to the function in which their storage is created, and each of these variables can only be used from within the

Program 7-10

```
#include <stdio.h>
int firstnum;     /* create a global variable named firstnum */
main( )
{
  int secnum;     /* create a local variable named secnum */
  void valfun(void);   /* function prototype */

  firstnum = 10; /* store a value into the global variable */
  secnum = 20;   /* store a value into the local variable  */

  printf("\nFrom main( ): firstnum = %d",firstnum);
  printf("\nFrom main( ): secnum = %d\n",secnum);
  valfun( );      /* call the function valfun */
  printf("\nFrom main( ) again: firstnum = %d",firstnum);
  printf("\nFrom main( ) again: secnum = %d",secnum);
}

void valfun(void)     /* no values are passed to this function */
{                     /* and no value is returned              */
  int secnum;  /* create a second local variable named secnum */

  secnum = 30; /* this only affects this local variable's value */
  printf("\nFrom valfun( ): firstnum = %d",firstnum);
  printf("\nFrom valfun( ): secnum = %d\n",secnum);
  firstnum = 40;     /* this changes firstnum for both functions */
  return;
}
```

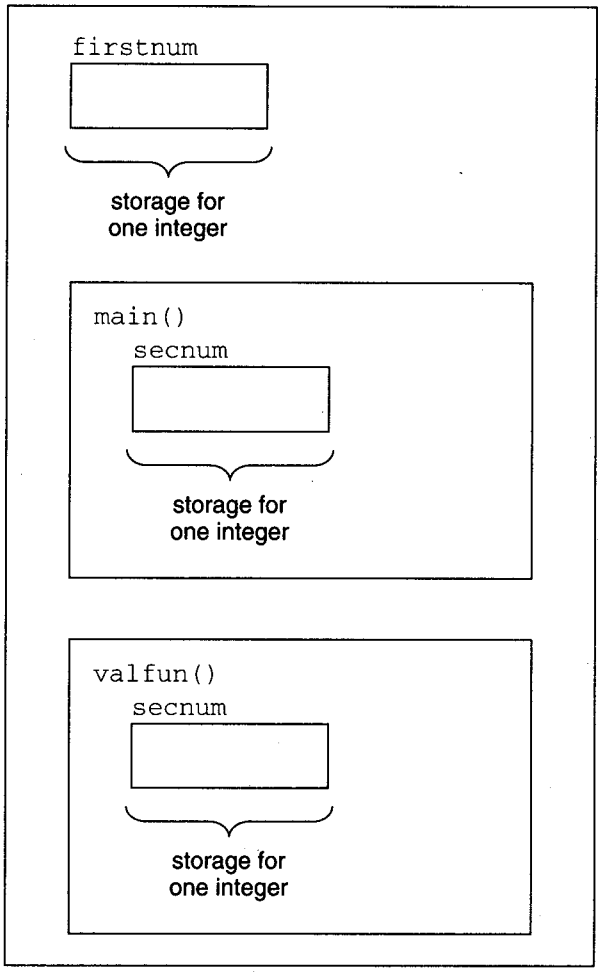


FIGURE 7–12 The Three Storage Areas Created by Program 7-10

appropriate function. Thus, when `secnum` is used in `main( )`, the storage area reserved by `main( )` for its `secnum` variable is accessed, and when `secnum` is used in `valfun( )`, the storage area reserved by `valfun( )` for its `secnum` variable is accessed. The following output is produced when Program 7-10 is run:

```
From main( ): firstnum = 10
From main( ): secnum = 20

From valfun( ): firstnum = 10
From valfun( ): secnum = 30

From main( ) again: firstnum = 40
From main( ) again: secnum = 20
```

Let us analyze the output produced by Program 7-10. Since `firstnum` is a global variable, both the `main( )` and `valfun( )` functions can use and change its value. Initially, both functions print the value of `10` that `main( )` stored in

`firstnum`. Before returning, `valfun( )` changes the value of `firstnum` to `40`, which is the value displayed when the variable `firstnum` is next displayed from within `main( )`.

Since each function only "knows" its own local variables and any defined global variables, `main( )` can only send the value of its `secnum` to the `printf( )` function, and `valfun( )` can only send the value of its `secnum` to the `printf( )` function. Thus, whenever `secnum` is obtained from `main( )` the value of `20` is displayed, and whenever `secnum` is obtained from `valfun( )` the value `30` is displayed.

C does not confuse the two `secnum` variables because only one function can execute at a given moment. Only the storage area for the variables created by the function currently being executed are accessed. If a variable that is not local to the function is used by the function, the program searches the global storage areas for the correct name. An error results if no variable is located.

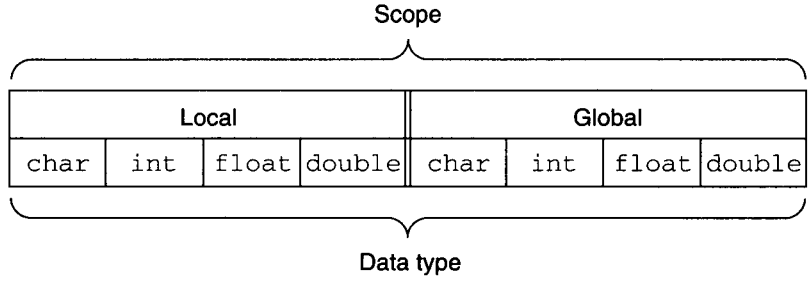
The scope of a variable in no way influences or restricts the data type of the variable. Just as a local variable can be a character, integer, float, double, or any of the other data types (long/short) we have introduced, so can global variables be of these data types, as illustrated in Figure 7–13. The scope of a variable is initially set by the placement of the declaration statement that reserves storage for it, while the data type of the variable is determined by using the appropriate keyword (`char`, `int`, `float`, `double`, etc.) before the variable's name in a declaration statement.

Misuse of Globals

One caution should be mentioned here. Global variables allow the programmer to "jump around" the normal safeguards provided by functions. Rather than passing variables to a function, it is possible to make all variables global ones. *Do not do this.* By indiscriminately making all variables global you instantly destroy the safeguards C provides to make functions independent and insulated from each other, including the necessity of carefully designating the type of arguments needed by a function, the variables used in the function, and the value returned.

Using only global variables can be especially disastrous in larger programs that have many user-created functions. Since all variables in a function must be declared, creating functions that use global variables requires that you remember to write the appropriate global declarations at the top of each program using the function—they no longer come along with the function. More troublesome than

FIGURE 7–13 Relating the Scope and Type of a Variable



this, however, is the horror of trying to track down an error in a large program that uses all global variables. Since a global variable can be accessed and changed by any function following the global declaration, it is a time-consuming and frustrating task to locate the origin of an erroneous value.

Global variables, however, are extremely useful in creating tables of data and constants that must be shared between many functions. If many functions require access to a group of tables, global variables allow the functions to make efficient changes to the same table without the need for multiple table passing.

Exercises 7.4

1 a. For the following section of code, determine the data type and scope of all declared variables. To do this use a separate sheet of paper and list the three column headings that follow (we have filled in the entries for the first variable):

Variable name	Data type	Scope
`price`	`integer`	global to `main( )`, `roi( )`, and `step( )`

```
int price;
long int years;
double yield;
main( )
{
  int bondtype;
  double interest, coupon;
     .
     .
}

double roi(int mat1, int mat2)
{
  int count;
  double eff_int;
       .
       .
  return(eff_int);
}

int step(float first, float last)
{
  int numofyrs;
  float fracpart;
       .
       .
  return(10*numofyrs);
}
```

b. Draw boxes around the appropriate section of the above code to enclose the scope of each variable.

c. Determine the data type of the arguments that the functions `roi( )` and `step( )` expect, and the data type of the value returned by these functions.

2 a. For the following section of code, determine the data type and scope of all declared variables. To do this use a separate sheet of paper and list the three column headings that follow (we have filled in the entries for the first variable):

Variable name	Data type	Scope
key	char	global to main( ), func1( ), and func2( )

```
char key;
long int number;
main( )
{
  int a,b,c;
  double x,y;
     .
     .
}

double secnum;

int func1(int num1, int num2)
{
  int o,p;
  float q;
     .
     .
  return(p);
}

double func2(float first, float last)
{
  int a,b,c,o,p;
  float r;
  double s,t,x;
     .
     .
  return(s*t);
}
```

b. Draw a box around the appropriate section of the above code to enclose the scope of the variables key, secnum, y, and r).

c. Determine the data type of the arguments that the functions func1( ) and func2( ) expect, and the data type of the value returned by these functions.

3. Besides speaking about the scope of a variable, we can also apply the term to the arguments declared within a function's header line. What do you think is the scope of all function arguments?

4. Determine the values displayed by each call to printf( ) in the following program:

```
int firstnum = 10;    /* declare and initialize a globalvariable */
main( )
{
  int firstnum = 20; /* declare and initialize a local variable */

  printf("\nThe value of firstnum is %d",firstnum);
  display( );
}

display( )
{
printf("\nThe value of firstnum is now %d",firstnum);
}
```

7.5 Variable Storage Class

The scope of a variable defines the location within a program where that variable can be used. Given a program, you could take a pencil and draw a box around the section of the program where each variable is valid. The space inside the box would represent the scope of a variable. From this viewpoint, the scope of a variable can be thought of as the space within the program where the variable is valid.

In addition to the space dimension represented by its scope, variables also have a time dimension. The time dimension refers to the length of time that storage locations are reserved for a variable. For example, all variable storage locations are released back to the computer when a program is finished running. However, while a program is still executing, interim variable storage areas are reserved and subsequently released back to the computer. Where and how long a variable's storage locations are kept before they are released can be determined by the storage class of the variable.

Besides having a data type and scope, every variable also has a storage class. The four available storage classes are called `auto`, `static`, `extern`, and `register`. If one of these class names is used, it must be placed before the variable's data type in a declaration statement. Examples of declaration statements that include a storage class designation are:

```
auto int num;       /* auto storage class and int data type  */
static int miles;   /* static storage class and int data type   */
register int dist; /* register storage class and int data type */
extern int price;   /* extern storage class and int data type  */
auto float coupon; /* auto storage class and float data type   */
static double yrs; /* static storage class and double data type */
extern float yld;   /* extern storage class and float data type  */
auto char in_key;   /* auto storage class and char variable      */
```

To understand what the storage class of a variable means, we will first consider local variables (those variables created inside a function) and then global variables (those variables created outside a function).

Local Variable Storage Classes

Local variables can only be members of the `auto`, `static`, or `register` storage classes. If no class description is included in the declaration statement, the variable is automatically assigned to the `auto` class. Thus, `auto` is the default class used by C for local variables. All the local variables we have used, since the storage class designation was omitted, have been `auto` variables.

The term `auto` is short for automatic. Storage for automatic local variables is automatically reserved or created each time a function declaring automatic variables is called. As long as the function has not returned control to its calling function, all automatic variables local to the function are "alive"—that is, storage for the variables is available. When the function returns control to its calling function, its local automatic variables "die"—that is, the storage for the variables is released back to the computer. This process repeats itself each time a function is called. For example, consider Program 7-11, where the function `testauto( )` is called three times from `main( )`.

Program 7-11

```
#include <stdio.h>
main( )
{
  int count;  /* create the auto variable count */
  void testauto(void);   /* function prototype */

  for(count = 1; count <= 3; ++count)
    testauto();
}

void testauto(void)    /* no value is passed to the function */
{                      /* and no value is returned           */
  int num = 0;  /* create the auto variable num */
                /* and initialize to zero       */

  printf("\nThe value of the automatic variable num is %d", num);
  ++num;
  return;
}
```

The output produced by Program 7-11 is:

```
The value of the automatic variable num is 0
The value of the automatic variable num is 0
The value of the automatic variable num is 0
```

Each time `testauto( )` is called, the automatic variable `num` is created and initialized to zero. When the function returns control to `main( )` the variable `num` is destroyed along with any value stored in `num`. Thus, the effect of incrementing `num` in `testauto( )`, before the function's `return` statement, is lost when control is returned to `main( )`.

For most applications, the use of automatic variables works just fine. In some cases, however, we would like a function to remember values between function calls. This is the purpose of the `static` storage class. A local variable that is declared as `static` causes the program to keep the variable and its latest value even when the function that declared it is through executing. Examples of `static` variable declarations are:

```
static int rate;
static float taxes;
static double amount;
static char in_key;
static long years;
```

A local `static` variable is not created and destroyed each time the function declaring the `static` variable is called. Once created, local `static` variables

remain in existence for the life of the program. This means that the last value stored in the variable when the function is finished executing is available to the function the next time it is called.

Since local `static` variables retain their values, they are not initialized within a declaration statement in the same way as automatic variables. To see why, consider the automatic declaration `int num = 0;`, which causes the automatic variable `num` to be created and set to zero each time the declaration is encountered. This is called a run-time initialization because initialization occurs each time the declaration statement is run. This type of initialization would be disastrous for a `static` variable, because resetting the variable's value to zero each time the function is called would destroy the very value we are trying to save.

The initialization of `static` variables (both local and global) is done only once, when the program is first compiled. At compilation time the variable is created and any initialization value is placed in it. Thereafter, the value in the variable is kept without further initialization each time the function is called. To see how this works, consider Program 7-12.

Program 7-12

```
#include <stdio.h>
main( )
{
  int count;                /* count is a local auto variable */
  void teststat(void);      /* function prototype */

  for(count = 1; count <= 3; ++count)
    teststat( );
}

void teststat(void)         /* no values are passed to the    */
{                           /* function and none are returned */
  static int num = 0;       /* num is a local static variable */

  printf("\nThe value of the static variable num is now %d", num);
  ++num;
  return;
}
```

The output produced by Program 7-12 is:

```
The value of the static variable num is now 0
The value of the static variable num is now 1
The value of the static variable num is now 2
```

As illustrated by the output of Program 7-12, the `static` variable `num` is set to zero only once. The function `teststat( )` then increments this variable just before returning control to `main( )`. The value that `num` has when leaving the

function `teststat( )` is retained and displayed when the function is next called.

Unlike automatic variables that can be initialized by either constants or expressions using both constants and previously initialized variables, `static` variables can only be initialized using constants or constant expressions, such as `3.2 + 8.0`. Also, unlike automatic variables, all `static` variables are set to zero when no explicit initialization is given. Thus, the specific initialization of `num` to zero in Program 7-12 is not required.

The remaining storage class available to local variables, the `register` class, is not used as extensively as either automatic or `static` variables. Examples of `register` variable declarations are:

```
register int time;
register double diffren;
register float coupon;
```

`Register` variables have the same time duration as automatic variables; that is, a local `register` variable is created when the function declaring it is entered, and is destroyed when the function completes execution. The only difference between `register` and automatic variables is where the storage for the variable is located.

Storage for all variables (local and global), except `register` variables, is reserved in the computer's memory area. Most computers have a few additional high-speed storage areas located directly in the computer's processing unit that can also be used for variable storage. These special high-speed storage areas are called registers. Since registers are physically located in the computer's processing unit, they can be accessed faster than the normal memory storage areas located in the computer's memory unit. Also, computer instructions that reference registers typically require less space than instructions that reference memory locations because there are fewer registers that can be accessed than there are memory locations.

For example, although the AT&T WE 32100 central processing unit has nine registers that can be used for local C program variables, it can be connected to memories that have more than four billion bytes. Most other computers have a similar set of user-accessible registers but millions of memory locations. When the compiler substitutes the location of a register for a variable during program compilation, less space in the instruction is needed than is required to address a memory having millions of locations.

Besides decreasing the size of a compiled C program, using `register` variables can also increase the execution speed of a C program, if the computer you are using supports this data type. Variables declared with the `register` storage class are automatically switched to the `auto` storage class if your computer does not support `register` variables or if the declared `register` variables exceed the computer's register capacity.

The only restriction in using the `register` storage class is that the address of a `register` variable, using the address operator `&`, cannot be taken. This is easily understood when you realize that registers do not have standard memory addresses.

Global Variable Classes

Global variables, also referred to as external variables, are created by declaration statements external to a function. By their nature, these externally defined vari-

ables do not come and go with the calling of any function. Once an external (global) variable is created, it exists until the program in which it is declared is finished executing. Thus, external variables cannot be declared as either `auto` or `register` variables that are created and destroyed as the program is executing. Global variables may, however, be declared as members of the `static` or `extern` storage classes. Examples of declaration statements including these two class descriptions are:

```
extern int sum;
extern double price;
static double yield;
```

The `static` and `extern` classes affect only the scope, not the time duration, of global variables. As with `static` local variables, all global variables are initialized to zero at compile time.

The purpose of the `extern` storage class is to extend the scope of a global variable beyond its normal boundaries. To understand this, we must first note the fact that all of the programs we have written so far have always been contained together in one file. Thus, when you have saved or retrieved programs you have only needed to give the computer a single name for your program. This is not required by C.

Larger programs typically consist of many functions that are stored in multiple files. An example of this is shown in Figure 7–14, where the three functions

FIGURE 7–14 A Program May Extend Beyond One File

file1

```
int price;
float yield;
static double coupon;
main( )
{
   func1 ( );
   func2 ( );
   func3 ( );
   func4 ( );
}
func1( )
{
    .
    .
    .
}
func2( )
{
    .
    .
    .
}
```

file2

```
double interest;
func3( )
{
    .
    .
    .
}
func4( )
{
    .
    .
    .
}
```

main( ), func1( ), and func2( ) are stored in one file and the two functions func3( ) and func4( ) are stored in a second file.

For the files illustrated in Figure 7–14, the global variables price, yield, and coupon declared in file1 can only be used by the functions main( ), func1( ), and func2( ) in this file. The single global variable interest declared in file2 can only be used by the functions func3( ) and func4( ) in file2.

Although the variable price has been created in file1, we may want to use it in file2. Placing the declaration statement extern int price; in file2, as shown in Figure 7–15, allows us to do this. Putting this statement at the top of file2 extends the scope of the variable price into file2 so that it may be used by both func3( ) and func4( ).

Similarly, placing the statement extern float yield; in func4( ) extends the scope of this global variable, created in file1, into func4( ). The scope of the global variable interest, created in file2, is extended into func1( ) and func2( ) by the declaration statement extern double interest; placed before func1( ). Notice that interest is not available to main( ).

A declaration statement that specifically contains the word extern is different from every other declaration statement in that it does not cause the creation of a new variable by reserving new storage for the variable. An extern declaration statement simply informs the compiler that the variable already exists and can now be used. The actual storage for the variable must be created somewhere else in the program using one and only one global declaration statement in which the word extern has not been used. Initialization of the global variable

FIGURE 7–15 Extending the Scope of a Global Variable

file1

```
int price;
float yield;
static double coupon;
main ( )
{
  func1 ( );
  func2 ( );
  func3 ( );
  func4 ( );
}
extern double interest;
func1( )
{
   .
   .
   .
}
func2 ( )
{
   .
   .
   .
}
```

file2

```
double interest;
extern int price;
func3( )
{
   .
   .
   .
}
func4 ( )
{
  extern float yield;
   .
   .
   .
}
```

can, of course, be made with the original declaration of the global variable. Initialization within an `extern` declaration statement is not allowed and will cause a compilation error.

The existence of the `extern` storage class is the reason we have been so careful to distinguish between the creation and declaration of a variable. Declaration statements containing the word `extern` do not create new storage areas; they only extend the scope of existing global variables.

The last global class, `static` global variables, is used to prevent the extension of a global variable into a second file. Global `static` variables are declared in the same way as local `static` variables, except that the declaration statement is placed outside any function.

The scope of a global `static` variable cannot be extended beyond the file in which it is declared. This provides a degree of privacy for `static` global variables. Since they are only "known" and can only be used in the file in which they are declared, other files cannot access or change their values. `static` global variables cannot be subsequently extended to a second file using an `extern` declaration statement. Trying to do so will result in a linking error.

Exercises 7.5

1 ***a.*** List the storage classes available to local variables.
b. List the storage classes available to global variables.

2. Describe the difference between a local `auto` variable and a local `static` variable.

3. What is the difference between the following functions:

```
init1( )
{
  static int yrs = 1;

  printf("\nThe value of yrs is %d", yrs);
  yrs = yrs + 2;
}


init2( )
{
  static int yrs;

  yrs = 1;
  printf("\nThe value of yrs is %d", yrs);
  yrs = yrs + 2;
}
```

4 ***a.*** Describe the difference between a `static` global variable and an `extern` global variable.
b. If a variable is declared with an `extern` storage class, what other declaration statement must be present somewhere in the program?

5. The declaration statement `static double years;` can be used to create either a local or global `static` variable. What determines the scope of the variable `years`?

6. For the function and variable declarations illustrated in Figure 7–16, place an `extern` declaration to individually accomplish the following:

file1

```
char choice;
int flag;
long date, time;
main( )
{
    .
    .
    .
}
double coupon;
price( )
{
    .
    .
    .
}
yield( )
{
    .
    .
    .
}
```

file2

```
char b_type;
double maturity
roi( )
{
    .
    .
    .
}
pduction( )
{
    .
    .
    .
}
bid( )
}
    .
    .
    .
}
```

FIGURE 7–16 Files for Exercise 6

a. Extend the scope of the global variable `choice` into all of `file2`.
b. Extend the scope of the global variable `flag` into function `pduction( )` only.
c. Extend the scope of the global variable `date` into `pduction( )` and `bid( )`.
d. Extend the scope of the global variable `date` into `roi( )` only.
e. Extend the scope of the global variable `coupon` into `roi( )` only.
f. Extend the scope of the global variable `b_type` into all of `file1`.
g. Extend the scope of the global variable `maturity` into both `price( )` and `yield( )`.

7.6 Common Programming Errors

An extremely common programming error related to functions is passing incorrect data types. The values passed to a function must correspond to the data types of the arguments declared within the function. The simplest way to verify that correct values have been received is to display all passed values within a function's body before any calculations are made. Once this verification has taken place, the display can be dispensed with.

Another common error occurs when the same variable is declared locally within both the calling and called functions. Even though the variable name is the same, a change to one local variable does not alter the value in the other local variable.

Related to this error is the error caused when a local variable has the same name as a global variable. Within the function declaring it, the use of the local

variable name only affects the local variable's contents. Thus, the value of the global variable can never be altered by the function.

Another common error is omitting the called function's prototype within the calling function. The called function must be alerted to the type of value that will be returned, and this information is provided by the function prototype. This declaration can only be omitted if the called function is physically placed in a program before its calling function or the called function returns an `integer` or `void` data type. The actual value returned by a function can be verified by displaying it both before and after it is returned.

The last two common errors are terminating a function's header line with a semicolon and forgetting to include the data type of a function's parameters.

7.7 Chapter Summary

1. A function is called by giving its name and passing any data to it in the parentheses following the name. If a variable is one of the arguments in a function call, the called function receives a copy of the variable's value.
2. The commonly used form of a user-written function is:

```
return-type function-declarator
{
   declarations;
   other statements;
}
```

   The first line of the function is called the *function header*. The opening and closing braces of the function and all statements between these braces constitute the function's *body*.

   The storage class of the function specified in the header is optional and can be either `static` or `extern`. If no storage class is specified it defaults to `extern`. The returned data type is, by default, an integer when no returned data type is specified. The function declarator must be included for each function and is of the form:

```
function-name(parameter list)
```

   The parameter list must include the names of all parameters and their data types.
3. A function's return-type is the data type of the value returned by the function. If no type is declared the function is assumed to return an integer value. If the function does not return a value it should be declared as a `void` type.[7]
4. Functions can directly return at most a single data-type value to their calling functions. This value is the value of the expression in the `return` statement.
5. Functions can be declared to all calling functions by means of a *function prototype*. The prototype provides a declaration for a function that specifies the

[7] Strictly speaking this includes the `main( )` function. By convention, however, `main( )` is typically written without declaring it as a `void` type.

data type returned by the function, its name, and the data types of the parameters expected by the function. As with all declarations, a function prototype is terminated with a semicolon and may be included within local variable declarations or as a global declaration. The most common form of a function prototype is:

```
data-type function-name(parameter data types);
```

If the called function is placed physically above the calling function no further declaration is required, since the function's definition serves as a global declaration to all following functions.

6. A set of preprogrammed functions for input, output, mathematical procedures, and string handling are included in the standard library provided with each C compiler. To use one of these functions you must obtain the name of the function, the arguments expected by the function, the data type of the returned value (if any), and a description of what the function does.
7. Every variable used in a program has a scope, which determines where in the program the variable can be used. The scope of a variable is either local or global and is determined by where the variable's definition statement is placed. A local variable is defined within a function and can only be used within its defining function or block. A global variable is defined outside a function and can be used in any function following the variable's definition. All global variables (which are formally called external variables) are initialized to zero and can be shared between files using the keyword `extern`.
8. Every variable has a class. The class of a variable determines how long the value in the variable will be retained. `auto` variables are local variables that exist only while their defining function is executing. `register` variables are similar to automatic variables but are stored in a computer's internal registers rather than in memory. `static` variables can be either global or local and retain their values for the duration of a program's execution. `static` variables are also set to zero when they are defined.
9. A function can also be passed the address of a variable. The storage of addresses is presented in Chapter 10. By passing addresses, a function has the capability of effectively returning many values.

7.8 Enrichment Study: Programming Costs

Any project that requires a computer incurs both hardware and software costs. The costs associated with the hardware consist of all costs relating to the physical components used in the system. These components include the computer itself, peripherals, and any other items, such as air conditioning, cabling, and associated equipment required by the project. The software costs include all costs associated with initial program development and subsequent program maintenance. As illustrated in Figure 7–17, the major cost of most engineering projects, be they research or development, has become the software costs.

The reason that software costs contribute so heavily to total project costs is that these costs are closely related to human productivity (labor intensive), while hardware costs are more directly related to manufacturing technologies. For

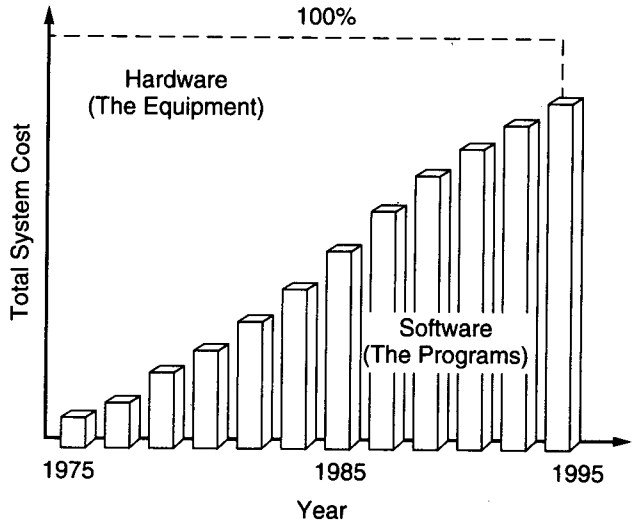


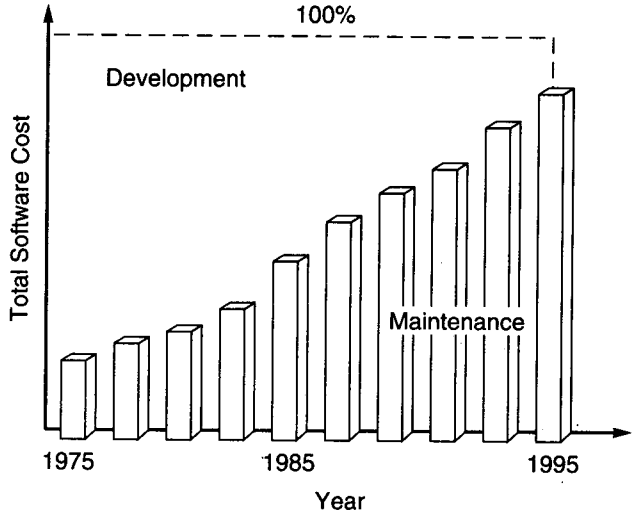
FIGURE 7–17 Software Is the Major Cost of Most Engineering Projects

example, microchips that cost over $500 per chip ten years ago can now be purchased for under $1 per chip.

It is far easier, however, to dramatically increase manufacturing productivity by a thousand, with the consequent decrease in hardware costs, than it is for people to double either the quantity or quality of their thought output. So as hardware costs have plummeted, software productivity and their associated costs have remained rather constant. Thus, the percentage of software costs to total system costs (hardware plus software) has increased dramatically.

Looking at just software costs (see Figure 7–18), we find that the maintenance of existing programs accounts for approximately 75 percent of total software costs. Maintenance includes the correction of newly found errors and the addition of new features and modifications to existing programs.

FIGURE 7–18 Maintenance Is the Predominant Software Cost



Students generally find it strange that maintenance is the predominant software cost because they are accustomed to solving a problem and moving on to a different one. Science and engineering fields do not operate this way. In these fields, one application or idea is typically built on a previous one and may require months or years of work. This is especially true in programming. Once a program is written, new features become evident. Advances in technology such as networking, fiber optics, genetic engineering, and graphical displays also open up new software possibilities.

How easily a program can be maintained (debugged or modified) is related to the ease with which the program can be read and understood, which is directly related to the modularity with which the program was constructed. Modular programs are constructed using one or more functions, each of which performs a clearly defined and specific task. If each function is clearly structured internally and the relationship between functions is clearly specified, each function can be tested and modified with a minimum of disturbance or undesirable interaction with the other functions in the program.

Just as hardware designers frequently locate the cause of a hardware problem by using test methods designed to isolate the offending hardware subsystem, modular software permits the software engineer to similarly isolate program errors to specific software functions.

Once a bug has been isolated or a new feature needs to be added, the required changes can be confined to appropriate functions without radically affecting other functions. Only if the affected function requires different input data or produces different outputs are its surrounding functions affected. Even in this case the changes to the surrounding functions are clear; they must either be modified to output the data needed by the changed function or changed to accept the new output data. Functions help the programmer determine where the changes must be made, while the internal structure of the function determines how easy it will be to make the change.

Although there are no hard and fast rules for well-written functions, specific guidelines do exist. The total number of instructions in a function generally should not exceed 50 lines. This allows the complete function to fit on a standard 8 1/2-by-11-inch sheet of paper for ease of reading. Each function ideally should have one entrance point and one exit point, and each control structure in the function (such as a `while` or `for` loop) should also contain a single entry and exit. This makes it easy to trace the flow of data when errors are detected. All the C statements that alter the normal sequential program flow, including the `if-else`, `while`, and `for` statements, conform to this single input–single output model.

As we have stressed throughout the text, the instructions contained within a module should use variable names that describe the data and are self-documenting. This means that they tell what is happening without a lot of extra comments. For example, the statement

```
x = (a - b) / (c - d);
```

does not contain intelligent variable names that give an indication of what is being calculated. A more useful set of instructions, assuming that a slope is being calculated, is:

```
slope = (y2 - y1) / (x2 - x1);
```

Here, the statement itself "tells" what the data represents, what is being calculated, and how the calculation is being performed. Always keep in mind that the goal is to produce programs that make sense to any programmer reading them, at any time. The use of mnemonic data names makes excessive comments unnecessary. The program should contain a sufficient number of comments explaining what a function does and any other pertinent information that would be helpful to other programmers; but excessive comments are usually a sign of insufficient program design or poorly constructed coding.

Another sign of a good program is the use of indentation to alert a reader to nested statements and indicate where one statement ends and another begins. Consider the pseudocode listed in the module "What to Wear" shown in Figure 7–19.

Because the `if` and `else` statement matchings are not clearly indicated, the instructions in the module are open to multiple interpretations. For example, using Figure 7–19, try to determine what to wear if the temperature is 35 degrees and it is raining. Now consider Version 2 of "What to Wear" in Figure 7–20.

Version 2 is indented, making it clear that we are dealing with one main `if-else` statement. If it is below 60 degrees the set of instructions indented underneath the first `if` will be executed; otherwise the condition `if it is raining` will be checked.

FIGURE 7–19 Version 1—What to Wear

```
if it is below 60 degrees
if it is snowing
wear your lined raincoat
else
wear a topcoat
if it is below 40 degrees
wear a sweater also
if it is below 30 degrees
wear a jacket also
else if it is raining
wear an unlined raincoat
```

FIGURE 7–20 Version 2—What to Wear

```
if it is below 60 degrees
   if it is snowing
      wear your lined raincoat
   else
      wear a topcoat
   if it is below 40 degrees
      wear a sweater also
      if it is below 30 degrees
         wear a jacket also
else if it is raining
   wear an unlined raincoat
```

Data Files

Chapter Eight

8.1 Creating and Using Data Files
8.2 Random File Access
8.3 Applications
8.4 Text and Binary Files
8.5 Common Programming Errors
8.6 Chapter Summary
8.7 Enrichment Study: Control Codes

The data for the programs we have seen so far either have been assigned internally within the programs or entered interactively during program execution. In this chapter we learn how to store data outside a program. This external data storage permits a program to use the data without having to recreate them each time the program is run. Additionally, it provides the basis for sharing data between programs, so that the data output by one program can be input directly to another program.

Any collection of data that is stored together under a common name on a storage medium other than the computer's main memory is called a *data file.* Typically data files are stored on floppy diskettes, hard disks, or magnetic tapes. This chapter describes the C statements needed to create data files, to write data to them, and to read the data from them.

8.1 Creating and Using Data Files

A data file is physically stored by a computer using a unique file name. Typically, most computers require that the file name consist of no more than eight characters followed by an optional period and an extension of up to three characters.[1] Using this convention, the following are all valid computer data file names:

```
math.dat    djavg.stk    records
infor.dat   exper1.dat   results.mem
```

Computer data file names should be chosen to indicate the file's information content. For data files the first eight characters typically are used to describe the data and the three characters after the decimal point are used to describe either the application or are set equal to `dat` to indicate a data file. For example, the file name `exper1.dat` is useful for describing a file of data pertaining to experiment number 1. Similarly, the file name `djavg.stk` could be used for Dow Jones averages required in a stock-related program.

Within a C program a file is always referenced by a variable name that must be declared within the program. The declaration takes the form

```
FILE *filename;
```

where the asterisk is required and `filename` is selected by the programmer and can be any valid variable name. Examples of valid file declarations are:

```
FILE *in_file;
FILE *factors;
FILE *weights;
FILE *out_file;
```

[1] Although this convention has been adopted by many computer systems (for example, VMX and DOS), the maximum number of characters allowed for a file name is system dependent.

Within a C program a file is always referenced using the variable (file) name declared in the FILE declaration statement.[2] The correspondence between this "internal" file name and the file's "external" computer name is made using a C function named fopen( ). Two arguments are required to use fopen( ). The first argument is the computer's name for the file; the second argument is the mode in which the file is to be used. The most commonly used modes are "r", "w", and "a", which represent reading, writing, or appending to a file, respectively.[3] For example, the statement

```
out_file = fopen("exper1.dat", "w");
```

opens a file named exper1.dat and assigns this to file name out_file. The name out_file is a programmer-selected name for the file that is declared in the FILE declaration and represents how the file is referenced within the C program that opened the file.

A file opened for writing creates a new file and makes the file available for use within the function opening the file. If a file exists with the same name as a file opened for writing, the old file is overwritten. Thus, the statement

```
out_file=fopen("exper1.dat","w");
```

opens a file named exper1.dat that can now be written to. Once this file has been opened, the program accesses the file using the name out_file, while the computer saves the file under the name exper1.dat.

A file opened for appending makes an existing file available for data to be added to the end of the file. If the file opened for appending does not exist, a new file with the designated name is created and made available to receive output from the program. For example, the statement

```
out_file = fopen("exper1.dat","a");
```

opens a file named exper1.dat and makes it available for data to be appended to the end of the file.

The only difference between a file opened in write mode and one opened in append mode is where the data are physically placed in the file. In write mode, the data are written starting at the beginning of the file, while in append mode the data are written starting at the end of the file. For a new file, the two modes are identical.

A file opened in read mode retrieves an existing file and makes its data available as input to the program. For example, the open statement

```
in_file = fopen("test.dat","r");
```

---

[2] The asterisk in the FILE declaration statement *does not* mean multiplication. An asterisk used in a declaration statement means the variable immediately following it is a pointer variable, which is explained in detail in Chapter 10.

[3] Additionally, some systems support the modes r+, w+, and a+. The r+ mode opens an existing file for reading and updating existing records; the w+ mode erases an existing file and opens a blank file for reading and writing; and the a+ mode allows reading, updating, and appending to a file.

opens the file named `test.dat` and makes the data in the file available for input. Within the function opening the file, the file is read using the name `in_file`.

As an executable statement, a call to the `fopen( )` function can be made anywhere within a function after its declaration statements. The function prototype for `fopen( )` is contained in the `stdio.h` header file.

When a call to `fopen( )` is encountered the computer checks whether the file currently exists on the system. If a file having the indicated file name exists, the file is opened. If the file does not exist, and the file was opened in write or append modes (`"w"` or `"a"`), a blank file having the indicated name is created. If a file opened for reading does not exist, the `fopen( )` function returns the system-named constant `NULL`. This named constant can be used to test that an existing file has, in fact, been opened.

Program 8-1 illustrates the statements required to open a file in read mode and the use of the returned value from `fopen( )` to check for a successful opening of the file.

Program 8-1

```
#include <stdio.h>
main( )
{
  FILE *in_file;

  in_file = fopen("test.dat","r");          /* open the file */

  if (in_file == NULL)  /* if fopen() returns NULL to in_file */
  {
    printf("\nThe file cannot be opened.");
    printf("\nPlease check that the file currently exists.");
  }
  else
    printf("\nThe file has been successfully opened for reading.");

}
```

When Program 8-1 is run on a system that does have the file named `test.dat` on it, the message

```
The file has been successfully opened for reading.
```

is displayed. If the file does not exist, however, the message

```
The file cannot be opened.
Please check that the file currently exists.
```

is displayed.

Although Program 8-1 can be used to open an existing file in read mode, it clearly lacks statements to read the data in the file and then close the file.

Writing to a File

If a file is opened in write mode, data can be written to it using almost identical functions for writing data to a display screen. For writing to a file these functions are listed in Table 8–1. The function prototypes for these functions are contained in `stdio.h`.

For example, if `out_file` is the file name assigned when the file was opened in either write or append modes, the following statements are valid:

```
futc('a',out_file);                /* write an a to the file */
fputs("Hello World!",out_file); /* write the string to the file */
fprintf(out_file,"%f %f %f", weight, factor, balance);
```

Notice that the `fprintf( )` file function is used in the same manner as the equivalent `printf( )` function, with the addition of the file name as an argument. The file name directs the output to a specific file instead of to the standard display device. Program 8-2 illustrates the use of an `fopen( )` function and two subsequent calls to `fprintf( )` for writing data to an opened file.

Program 8-2

```
#include <stdio.h>
main( )
{
  FILE *out_file;          /* FILE declaration */
  float weight = 165.0 , slope = 7.5, factor = 2.0625;

  out_file = fopen("test.dat","w");
  fprintf(out_file,"%f",weight);
  fprintf(out_file,"\n%f   %f",slope, factor);
}
```

TABLE 8–1 File Writing Functions

Function	Description
`fputc(c,filename)`	Write a single character to the file.
`fputs(string,filename)`	Write a string to the file.
`fprintf(filename,"format",args)`	Write the values ofthe arguments to the file according to the format.

When Program 8-2 is executed, a file named `test.dat` is created by the computer. After the file is opened two `fprintf( )` function call statements are used to write two lines to the `test.dat` file. Formally, each line in a file is referred to as a *record*. Thus, the file produced by Program 8-2 consists of the following two records:

```
165.000000
7.500000    2.062500
```

As illustrated in Program 8-2, writing to a file is essentially the same as writing to the standard output device, except for the explicit designation of the file's name and the use of `fprintf( )` in place of `printf( )`. This means that all the techniques you have learned for creating standard output displays apply to file writes as well. For example, Program 8-3 illustrates storing data from an array into a file opened as `newfile`.

Program 8-3

```
#include <stdio.h>
main( )
{
  FILE *newfile;
  int i;
  float result[5] = {16.25, 17.0, 15.75, 18.0, 19.5};

  newfile = fopen("exper.dat","w");  /* open the file */
  for (i = 0; i < 5; ++i)
    fprintf(newfile,"\n%1d  %9.6f", i, result[i]);
  fclose(newfile);
}
```

When Program 8-3 is executed, a file named `exper.dat` is opened by the computer (if the file does not exist it is automatically created). After the file is opened a `for` loop is used to write five lines to the file, with each line containing two items. The file produced by this program consists of the following five lines:

```
0  16.250000
1  17.000000
2  15.750000
3  18.000000
4  19.500000
```

Closing a File

Included in Program 8-3 is a call to the `fclose( )` function. This function is used to formally break the link established by the `fopen( )` function call and

release the internal file name, which can then be used for another file. The general form of the call to `fclose( )` is:

```
fclose(filename);
```

where `filename` is the "internal" name of the file used when the file was opened.

Since all computers have a limit on the maximum number of files that can be open at one time, closing files that are no longer needed makes good sense. In the absence of a specific `fclose( )` function call, as in Program 8-2, any open files existing at the end of normal program execution are automatically closed by the operating system.

When a file is closed a special end-of-file (`EOF`) marker is automatically placed by the operating system as the last character in the file. The `EOF` character has a unique numerical code that has no equivalent representation as a printable character. This special numerical value, which is system dependent, ensures that the `EOF` character can never be confused with a valid character contained internally within the file. As we will see shortly, this `EOF` character can be used as a sentinel when reading data from a file.

Reading a File

Reading data from a file is almost identical to reading data from the keyboard. The functions that are used for reading file data are listed in Table 8–2. The function prototypes for these functions are contained in `stdio.h`.

For example, if `in_file` is the name of a file opened in read mode, the following statements could be used to read data from the file:

```
fgetc(in_file);          /* read the next character in the file */

fgets(message,10,in_file); /* read the next 9 characters from */
                           /* the file into message           */

fscanf(in_file,"%f",&factor); /* read a floating point number */
```

All the input functions correctly detect the end-of-file marker. The functions `fgetc( )` and `fscanf( )`, however, return the named constant `EOF` when the

TABLE 8–2 File Reading Functions

Function	Description
`fgetc(filename)`	Read a character from the file.
`fgets(stringname,n,filename)`	Read *n*-1 characters from the file and store the characters in the given string name.
`fscanf(filename,"format",&args)`	Read values for the listed arguments from the file according to the format.

marker is detected. The function `fgets( )` returns a NULL (\0) when it detects the end of a file. Both of these symbolic constants, EOF and NULL, are useful when reading a file to detect when the end of the file has been reached.

Reading data from a file requires that the programmer knows how the data appears in the file. This is necessary for correct "stripping" of the data from the file into appropriate variables for storage. All files are read sequentially, so that once an item is read the next item in the file becomes available for reading. Program 8-4 uses the `fscanf( )` function to read five lines from the `exper.dat` file written by Program 8-3. Each time the file is read, an integer and a real value are input to the program and displayed.

Program 8-4

```
#include <stdio.h>
main( )
{
  FILE *in_file;
  int i, n;
  float val;

  in_file = fopen("exper.dat","r");
  for (i = 1; i <=5; ++i)
  {
    fscanf(in_file,"%d %f", &n, &val);
    printf("\n%d  %f",n,val);
  }
  fclose(in_file);
}
```

The display produced by Program 8-4 is:

```
0  16.250000
1  17.000000
2  15.750000
3  18.000000
4  19.500000
```

In addition to using a `for` loop to read a specific number of lines, as is done in Program 8-4, the end-of-file (EOF) marker appended to each file can be used as sentinel value. When the EOF marker is used in this manner, the following algorithm can be used to read and display each line of the file:

while not end-of-file
read a line

Program 8-5 illustrates reading the `exper.dat` file that was created in Program 8-3 using the EOF marker, which is returned by `fscanf( )` when the end of the file is encountered.

Program 8-5

```
#include <stdio.h>
main()
{
  FILE *in_file;
  int i, n;
  float val;

  in_file = fopen("exper.dat","r");
  while (fscanf(in_file,"%d %f", &n, &val) != EOF)
    printf("\n%d  %f",n,val);
  fclose(in_file);
}
```

Program 8-5 continues to read the file until the EOF marker has been detected. Each time the file is read, an integer and floating point number are input to the program. The display produced by Program 8-5 is:

```
0  16.250000
1  17.000000
2  15.750000
3  18.000000
4  19.500000
```

In place of the `fscanf( )` function used in Program 8-5, an `fgets( )` function call can be used. `fgets( )` requires three arguments: an address where the first character read will be stored, the maximum number of characters to be read, and the name of the input file. For example, the function call

```
fgets(&line[0],81,in_file);
```

causes a maximum of 80 characters (one less than the specified number) to be read from the file named `in_file` and stored starting at the address of element `line[0]` (recall from Section 3.2 that the ampersand symbol, `&`, mean "the address of"). `fgets( )` continues reading characters until 80 characters have been read or a newline character has been encountered. If a newline character is encountered it is included with the other entered characters before the string is terminated with the end-of-string marker, `\0`. `fgets( )` also detects the end-of-file marker, but returns the `NULL` character when the end of the file is encountered. Program 8-6 (p. 332) illustrates the use of `fgets( )` in a working program.

Program 8-6 is really a line-by-line text-copying program, reading a line of text from the file and then displaying it on the terminal. Thus, the output of Program 8-6 is identical to the output of Program 8-5. If it were necessary to obtain the integer and floating point numbers as individual variables, either Program 8-5 should be used or the string returned by `fgets( )` in Program 8-6 must be processed further using the string scan function, `sscanf( )`. For example, the statement

Program 8-6

```
#include <stdio.h>
main( )
{
  FILE *in_file;
  int i, n;
  float val;
  char line[81];

  in_file = fopen("exper.dat","r");
  while (fgets(&line[0], 81, in_file) != NULL)
    printf("%s",line);
  fclose(in_file);
}
```

```
sscanf(&line[0],"%d %f", &n, &val);
```

could be used to extract the description and price from the string stored in the `line` character array (see Section 11.4 for a more complete description of in-memory string formatting).

Standard Device Files

The data file names we have used have all been logical file names. A *logical file name* is one that references a file of related data that have been saved under a common name; that is, a data file. In addition to logical file names, C also supports physical file names. A *physical file name* refers to a hardware device, such as a keyboard, screen, or printer.

The actual physical device assigned to your program for data entry is formally called the standard input file. Usually this is a keyboard. When a `scanf( )` function call is encountered in a C program, the computer automatically goes to this standard input file for the expected input. Similarly, when a `printf( )` function call is encountered, the output is automatically displayed or "written to" a device that has been assigned as the standard output file. For most systems this is a CRT screen, although it can be a printer.

When a program is run, the keyboard used for entering data is automatically opened and assigned the internal file name `stdin`. Similarly, the output device used for display is assigned to the file name `stdout`. These file names are always available for programmer use.

The similarities between `printf( )` and `fprintf( )`, `scanf( )` and `fscanf( )` are not accidental. `printf( )` is a special case of `fprintf( )` that defaults to the standard output file, and `scanf( )` is a special case of `fscanf( )` that defaults to the standard input file. Thus,

```
fprintf(stdout,"Hello World!");
```

causes the same display as the statement

```
printf("Hello World!");
```

and

```
fscanf(stdin,"%d",&num);
```

is equivalent to the statement

```
scanf("%d",&num);
```

In addition to the `stdin` and `stdout` file names, a third file named `stderr` is assigned to the output device used for system error messages. Although `stderr` and `stdout` frequently refer to the same device, the use of `stderr` provides a means of redirecting any error messages away from the file being used for normal program output.

Just as `scanf( )` and `printf( )` are special cases of `fscanf( )` and `fprintf( )`, respectively, `getchar( )`, `gets( )`, `putchar( )`, and `puts( )` are also special cases of the more general file functions listed in Table 8–3.[4]

The character function pairs listed in Table 8–3 can be used as direct replacements for each other. This is not true for the string-handling functions. The differences between the string-handling functions are described as follows.

At input, as previously noted, the `fgets( )` function reads data from a file until a newline escape sequence or a specified number of characters has been read. If `fgets( )` encounters a newline escape sequence, as we saw in Program 8-6, it is stored with the other characters entered. The `gets( )` function, however, does not store the newline escape sequence in the final string. Both functions terminate the entered characters with an end-of-string `NULL` character.

At output, both `puts( )` and `fputs( )` write all the characters in the string except for the terminating end-of-string `NULL`. `puts( )`, however, automatically adds a newline escape sequence at the end of the transmitted characters while `fputs( )` does not.

TABLE 8–3 Correspondence Between Selected I/O Functions

Function	General Form
`putchar(character)`	`fputc(character,stdout)`
`puts(string)`	`fputs(string,stdout)`
`getchar( )`	`fgetc(stdin)`
`gets(stringname)`	`fgets(stringname,n,stdin)`

[4] All of the routines on the left-hand side of Table 8–3 are defined in the header file <stdio.h> using the equivalent functions on the right-hand side of the table. Strictly speaking, the functions on the right are true C functions, while those on the left are macros. Macros are described in detail in Chapter 14.

Other Devices

The keyboard, display, and error-reporting devices are automatically opened and assigned the internal file names `stdin`, `stdout`, and `stderr`, respectively, whenever a C program begins execution. Additionally, other devices can be used for input or output if the name assigned by the system is known. For example, most IBM or IBM-compatible personal computers assign the name `prn` to the printer connected to the computer. For these computers, the statement `fprintf("prn","Hello World!");` causes the string `Hello World!` to be printed directly at the printer. Here `prn` must be enclosed in quotes because it is not a variable name.

Exercises 8.1

1. Using the reference manuals provided with your computer's operating system, determine:

a. the maximum number of characters that can be used to name a file for storage by the computer system

b. the maximum number of data files that can be open at the same time

2. Would it be appropriate to call a saved C program a file? Why or why not?

3. Write individual `fopen( )` function calls to link the following "external" data file names to the corresponding "input" file names. Open each file for writing:

External Name	Internal Name
`math.dat`	`out_file`
`book.dat`	`book`
`resist.dat`	`resfile`
`exper2.dat`	`exfile`
`prices.dat`	`pfile`
`rates.mem`	`ratefile`

4. Write `fclose( )` function calls for each of the files opened in Exercise 3.

5a. Write a C program that stores the following numbers into a file named `result.dat`: 16.25, 18.96, 22.34, 18.94, 17.42, 22.63.

b. Write a C program to read the data in the `result.dat` file created in Exercise 5a and display the data. Additionally, your program should compute and display the sum and average of the data. Check the sum and average displayed by your program using a hand calculation.

6a. Write a C program that prompts the user to enter five numbers. As each number is entered the program should write the number into a file named `user.dat`.

b. Write a C program that reads the data in the `user.dat` file created in Exercise 6a and displays each individual data item.

7a. Create a file containing the following car numbers, number of miles driven, and number of gallons of gas used by each car:

Car No.	Miles Driven	Gallons Used
54	250	19
62	525	38
71	123	6
85	1,322	86
97	235	14

b. Write a C program that reads the data in the file created in Exercise 7a and displays the car number, miles driven, gallons used, and the miles per gallon for each car. The output should also display the total miles driven, total gallons used, and average miles per gallon for all the cars. These totals should be displayed at the end of the output report.

8a. Create a file with the following data containing the part number, opening balance, number of items sold, and minimum stock required:

Part Number	Initial Amount	Quantity Sold	Minimum Amount
310	95	47	50
145	320	162	20
514	34	20	25
212	163	150	160

b. Write a C program to create an inventory report based on the data in the file created in Exercise 8a. The display should consist of the part number, current balance, and the amount that is necessary to bring the inventory to the minimum level.

9a. Create a file containing the following data:

Identification Number	Rate	Hours
10031	6.00	40
10067	5.00	48
10083	6.50	35
10095	8.00	50

b. Write a C program that uses the information contained in the file created in Exercise 9a to produce the following pay report:

```
ID No.     Rate    Hours    Regular Pay    Overtime Pay    Gross Pay
-------------------------------------------------------------------
```

Any hours worked above 40 hours are paid at time and a half. At the end of the individual output for each ID number, the program should display the totals of the regular, overtime, and gross pay columns.

10a. Store the following data in a file:

```
5 96 87 78 93 21 4 92 82 85 87 6 72 69 85 75 81 73
```

b. Write a C program to calculate and display the average of each group of numbers in the file created in Exercise 10a. The data are arranged in the file so that each group of numbers is preceded by the number of data items in the group. Thus, the first number in the file, 5, indicates that the next five numbers should be grouped together. The number 4 indicates that the following four numbers are a group, and the 6 indicates that the last six numbers are a group. (*Hint:* Use a nested loop. The outer loop should be executed three times.)

11. Rotech Systems is a distributor of high-speed memory devices for specialized computer applications. Each memory device in stock is stored by its tolerance. Lower tolerance devices are sold at a premium and used for more critical applications that require a tighter tolerance. Having just completed an annual check of inventory in stock, Rotech has found it has the following quantities of memory devices in stock:

Device Number	5% Tolerance	2% Tolerance	1% Tolerance
4016	464	612	129
4314	742	1,215	375
4311	517	820	298
4364	684	105	22
4464	771	200	358

Based on these data Rotech wants a report of how many devices should be ordered to ensure that it has at least 800 of each item in stock. Your first task is to create a file containing these inventory data. Each line in the file should consist of a device number and the three inventory levels for that part number. When the file has been created use it in a program that reads the data into a two-dimensional array, searches the array, and prints a report listing the amount of each part that must be ordered.

12a. Write a C program that uses either the random number generator described in Section 7.3 or one supplied by your computer system to select 1000 random numbers having values between 1 and 100. As each number is selected it should be written to a file called `number`.

b. Using the `number` file created in Exercise 12a, write a C program that reads the data in the file, computes the average of the 1000 data items, and writes a new file consisting of all values that are 10% above or below the calculated average.

13. Instead of using an actual file name in an `fopen( )` function call, a character variable can be used instead. For example, the statement

```
in_file = fopen(fname,"r");
```

equates the internal file name `in_file` to the file name assigned to the variable `fname`. Here the variable `fname` must be declared as a character variable of sufficient length to hold a valid file name. The following code illustrates how this `fopen( )` statement could be used in practice:

```
#include <stdio.h>
main( )
```

```
{
  FILE *in_file;
  char fname[13];

  printf("Enter a file name: ");
  gets(fname);
  in_file = fopen(fname,"r");
     .
     .
     .
}
```

The variable declaration statement in this code creates a character variable named `fname`. The code then requests that the file name be entered by the user. The entered name is stored in the character variable `fname`, which then is used as an argument to the `fopen( )` function. Using this code rewrite Program 8-3 so that the name of the data file is entered when the program is executed.

8.2 Random File Access

File organization refers to the way data are stored in a file. All the files we have used have *sequential organization*. This means that the characters in the file are stored in a sequential manner, one after another. Additionally, we have read the file in a sequential manner. The way data are retrieved from the file is called *file access*. The fact that the characters in the file are stored sequentially, however, does not force us to access the file sequentially.

The standard library functions `rewind( )`, `fseek( )`, and `ftell( )` can be used to provide random access to a file. In random access any character in the file can be read immediately, without first having to read all the characters stored before it.

The `rewind( )` function resets the current position to the start of the file. `rewind( )` requires the file name as its only argument. For example, the statement

```
rewind(in_file);
```

resets the file so that the next character accessed will be the first character in the file. A `rewind( )` is done automatically when a file is opened in read mode.

The `fseek( )` function allows the programmer to move to any position in the file. In order to understand this function, you must first clearly understand how data are referenced in the file.

Each character in a data file is located by its position in the file. The first character in the file is located at position 0, the next character at position 1, and so on. A character's position is also referred to as its offset from the start of the file. Thus, the first character has a 0 offset, the second character has an offset of 1, and so on for each character in the file.

The `fseek( )` function requires three arguments: the name of the file; the offset, as a long integer; and the position from which the offset is to be calculated. The general form of `fseek( )` is:

```
fseek(file_name, offset, origin)
```

The values of the origin argument can be either 0, 1, or 2. An origin of 0 means the offset is relative to the start of the file. An origin of 1 means that the offset is relative to the current position in the file, and an origin of 2 means the offset is relative to the end of the file. A positive offset means move forward in the file and a negative offset means move backward. Examples of `fseek( )` are:

```
fseek(in_file,4L,0);    /* go to the fifth character in the file */
fseek(in_file,4L,1);    /* move ahead five characters */
fseek(in_file,-4L,1);   /* move back five characters */
fseek(in_file,0L,0);    /* go to start of file - same as rewind() */
fseek(in_file,0L,2);    /* go to end of file */
fseek(in-file,-10L,2); /* go to 10 characters before the file's end */
```

In these examples, `in_file` is the name of the file used when the data file was opened. Notice that the offset passed to `fseek( )` must be a long integer; the appended `L` tells the compiler to consider the offset as such.

Program 8-7

```
#include <stdio.h>
main( )
{
int ch, n;
long int offset, last, ftell();
FILE *in_file;

in_file = fopen("temp.dat","r");
fseek(in_file,0L,2);       /* move to the end of the file */
last = ftell(in_file);    /* save the offset of the last character */
for(offset = 0; offset <= last; ++offset)
{
  fseek(in_file, -offset, 2);    /* position to the next character */
  ch = getc(in_file);            /* get the character */
  switch(ch)
  {
    case '\n': printf("LF : ");
               break;
    case EOF : printf("EOF : ");
               break;
    default  : printf("%c : ",ch);

               break;
  }
}
fclose(in_file);
}
```

The last function, `ftell( )`, simply returns the offset value of the next character that will be read or written. For example, if 10 characters have already been read from a file named `in_file`, the function call

```
ftell(in_file);
```

returns the long integer 10. This means that the next character to be read is offset 10 byte positions from the start of the file and is the eleventh character in the file.

Program 8-7 illustrates the use of `fseek( )` and `ftell( )` to read a file in reverse order, from last character to first. Each character is also displayed as it is read.

Assuming the file `temp.dat` contains the following data,

```
Bulbs   3.12
```

the output of Program 8-7 is:

```
EOF : 2 : 1 : . : 3 :  :  :  : s : b : l : u : B :
```

Program 8-7 initially goes to the last character in the file. The offset of this character, which is the end-of-file character, is saved in the variable `last`. Since `ftell( )` returns a long integer, `last` has been declared as a long integer. The function prototype for `ftell( )` is contained in `stdio.h`.

Starting from the end of the file, `fseek( )` is used to position the next character to be read, referenced from the back of the file. As each character is read, the character is displayed and the offset adjusted in order to access the next character.

Exercises 8.2

1. Determine the value of the offset returned by `ftell( )` in Program 8-7. Assume that the file `temp.dat` contains the data

```
Bulbs   3.12
```

2. Rewrite Program 8-7 so that the origin for the `fseek( )` function used in the `for` loop is the start of the file rather than the end. The program should still print the file in reverse order.

3. The function `fseek( )` returns `0` if the position specified has been reached, or `1` if the position specified was beyond the file's boundaries. Modify Program 8-7 to display an error message if `fseek( )` returns `1`.

4. Write a program that will read and display every second character in a file named `temp.dat`.

5. Using the `fseek( )` and `ftell( )` functions, write a C program that returns the total number of characters in a file. The name of the file should be a user-entered item.

6 Write a C program that reads and displays n characters starting from any position in a file. The program should accept three user-entered items: the name of the file, the offset of the first character to be read, and the number of characters to be read.

7. Assume that a data file consisting of a group of individual lines has been created. Write a C program that will read and display any desired user-entered line number of the file.

8.3 Applications

Formally, each line in a file is referred to as a *record*, where each record consists of one or more data items. Once a data file has been created, the majority of applications are concerned with updating the file's records to maintain currently accurate data.

In this section two such applications are presented. The first application uses a file as a data base for storing the 10 most recent pollen counts, which are used in the summer as allergy "irritability" measures. As a new reading is obtained it is added to the file and the oldest stored reading is deleted.

The second application presents an expanded file update procedure. In this application a file containing inventory data, consisting of book identification numbers and quantities in stock, is updated by information contained in a second file. This application requires that identification numbers in the two files be matched before a record is updated.

Application 1: Pollen Counts

Pollen count readings, which are taken from August through September in the northeastern region of the United States, measure the number of ragweed pollen grains in the air. Pollen counts in the range of 10 to 200 grains per cubic meter of air are typical during this time of year. Typically, pollen counts above 10 begin to affect a small percentage of hay fever sufferers, counts in the range of 30 to 40 noticeably bother approximately 30 percent of hay fever sufferers, and counts between 40 and 50 adversely affect over 60 percent of all hay fever sufferers.

A program is to be written that updates a file containing the 10 most recent pollen counts. As a new count is obtained it is to be added to the end of the file and the oldest count deleted from the file.[5] Additionally, the averages of the old and new files' data are calculated and displayed. For purposes of illustration, assume that a file named `pollen` containing the data shown in Figure 8–1 has already been created.

The pseudocode for the file update program is:

```
Display a message indicating what the program does
Request the name of the data file
Request a new pollen count reading
Open the data file
For ten data items
   Read a value into an array
   Add the value to a total
Endfor
Calculate and display the old ten-day average
Calculate and display the new ten-day average
Rewind the data file
Write the nine most recent pollen counts from the array to the file
Write the new pollen count to the file
Close the file.
```

[5] This type of data storage is formally referred to as a First-In First-Out (FIFO) list, which is also called a *queue*. If the list is maintained in Last-In First-Out order it is called a *stack*.

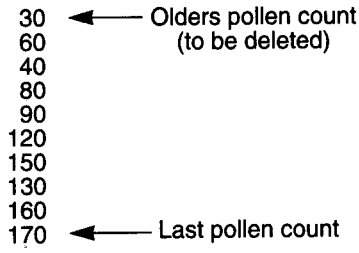


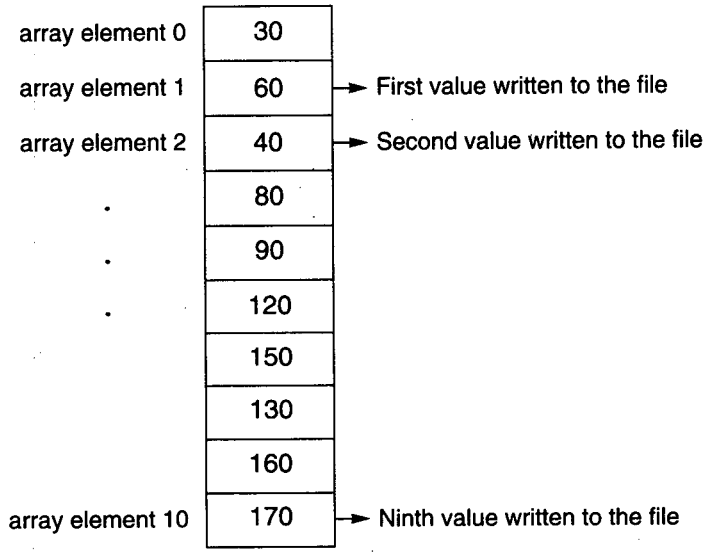
FIGURE 8–1 Data Currently in the Pollen File

In reviewing this algorithm, notice that an array is used for temporarily storing the contents of the file. Thus, when the file is initially read, the first pollen count in the file is stored in array element 0, the second element in array element 1, and so on, as illustrated in Figure 8–2. Once the data are "captured" in the array the data file can be rewound and written over with the latest 10 counts. As further illustrated in Figure 8–2 the first 9 values written to the file are taken from array elements 2 through 10, respectively. Finally, the last count written to the file is the most recent value. Program 8-8 expresses this algorithm in C.

Following is a sample run using Program 8-8:

```
This program updates the pollen count file
    and calculates ten count averages

Enter the pollen count file name: pollen
Enter the latest pollen count reading: 200

The old ten count average is: 103.00
The new ten count average is: 120.00

An updated data file has been written.
```

FIGURE 8–2 The Update Process



Program 8-8

```
#define POLNUMS 10
#include <stdio.h>
main( )
{
  int newcount, i, a[POLNUMS];
  float sum, average;
  char fname[13];
  FILE *pollen;

  /* get the data file name and most recent pollen count */
  printf("\nThis program updates the pollen count file");
  printf("\n    and calculates ten count averages");
  printf("\n\nEnter the pollen count file name: ");
  gets(fname);
  printf("Enter the latest pollen count reading: ");
  scanf("%d", &newcount);
  /* open the file, read, sum, and average the existing data */
  pollen = fopen(fname,"r+");
  sum = 0;
  for(i = 0; i < POLNUMS; ++i)
  {
    fscanf(pollen,"%d",&a[i]);
    sum = sum + a[i];
  }
  /* compute and display old and new averages */
  average = sum / (float) POLNUMS;
  printf("\nThe old ten count average is: %5.2f",average);
  sum = sum - a[0] + newcount;     /* update the sum */
  average = sum / (float) POLNUMS; /* calculate new average */
  printf("\nThe new ten count average is: %5.2f",average);
  /* write updated data to the file */
  rewind(pollen);
  for (i = 1; i < POLNUMS; ++i)
    fprintf(pollen,"%d ",a[i+1]);
  fprintf(pollen,"%d",newcount);
  fclose(pollen);
  printf("\n\nAn updated data file has been written.");
}
```

The updated file created by Program 8-8 is illustrated in Figure 8–3. In reviewing the contents of this file notice that the most current reading has been added to the end of the file and that the other counts are obtained from the original file shown in Figure 8–2, but moved up one position in the file.

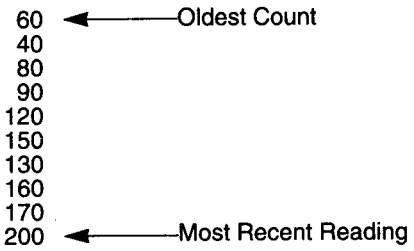


FIGURE 8–3 The Updated Pollen File

Application 2: Master/Transaction File Update

A common form of file update occurs when the update data are also contained in a file. Here the file to be updated is referred to as a *master file* and the file containing the update data is referred to as a *transactions file.*

As a specific example of this type of update, assume that the current master file named `oldbook.mas` contains the book identification numbers and quantities in stock illustrated in Table 8–4.

A transactions file named `book.trn` contains the quantities of each book bought, sold, or returned to stock each day. For purposes of illustration, assume that the `book.trn` file is sorted by ID number at the end of each month and contains the data illustrated in Table 8–5.

A standard solution to this problem is to have all the transactions kept in the same identification number order as the records in the master file. Since the

TABLE 8–4 Data Contained in the `oldbook.mas` File

Book ID No.	Quantity in Stock
125	98
289	222
341	675
467	152
589	34
622	125

TABLE 8–5 Data Contained in the Transaction File Named `book.trn`

ID No.	Date	Sold	Returned	Bought
289	1/10/92	125	34	50
341	1/10/92	300	52	0
467	1/15/92	50	20	200
467	1/20/92	225	0	160
589	1/31/92	75	10	55

records in the master file, as illustrated in Table 8–4, are in increasing (ascending) identification number order, the transactions must also be kept in ascending order. As illustrated in Table 8–5, this is the case for the `book.trn` file.

Once the two files are in the same ID number order the procedure for creating an updated master file consists of reading a master record from the existing master file and one record from the transaction file. If the ID numbers of the two records match, the transaction record's information is applied to the data in the master record and another transaction record is read. As long as the transaction record's ID number matches the master record's ID number, the update of the master record continues. When the transaction record's ID number does not match the master record's ID number, which indicates that there is no further update data to be applied to the master record, an updated record is written to the new master file. Let's see how this procedure works with the two files shown in Tables 8–4 and 8–5.

The first record read from the master file has ID number 125, while the first transaction record has ID number 289. Since the ID numbers do not match, the update of this first master record is complete (in this case there is no update information) and the existing master record is written without modification to the new master file. Then the next master record is read, which has an ID number of 289. Since this ID number matches the transaction ID number, the inventory balance for book number 289 is updated, yielding a new balance of 181 books. Because the transaction file can contain multiple update records for the same ID number (notice the two records for ID number 467) the next transaction record is read and checked before writing an updated record to the new master file. Since the ID number of the next transaction record is not 289, the update of this book number is complete and an updated master record is written to the new master file.

This algorithm continues, record by record, until the last master record has been updated. Should the end of the transaction file be encountered before the last master record is read from the existing master file, the remaining records in the existing master file are written directly to the new master file with no need to check for update information. The new master file can either be a completely new file, or each updated record can be written back to the old master file. In our update procedure we will create a new master file so that the original data in the old master file will be left intact.

Since we will be using two master files, the old and new masters, a notation must be established to clearly distinguish between them. By convention, the existing master file is always referred to as the old master file and the updated master file is called the new master file. Using these terms, the pseudocode description of the update procedure is:

```
open the old master file
open the new master file (initially blank)
open the transaction file
read the first old master record
while not at the end of the transaction file
   read a transaction record
   while the transaction ID does not match the old master ID
      write an updated master record to the new master file
      read the next old master record
   endwhile
   if the ID numbers do match
      calculate a new balance
```

```
        endwhile
/*      to get here the last transaction record has just been read    */
        write the last updated master to the new master file
        while there are any remaining records in the old master file
          read an old master record
          write a new master record
        endwhile
        close all files.
```

In C, this update procedure is described by Program 8-9.

Program 8-9

```
#include <stdio.h>
main( )
{
  int idmast, idtrans, balance, sold, returned, bought;
  char date[8];
  FILE *oldmast, *newmast, *trans;

  oldmast = fopen("oldbook.mas","r");
  newmast = fopen("newbook.mas","w");
  trans = fopen("book.trn","r");
  fscanf(oldmast,"%d %d", &idmast, &balance);
  while( fscanf(trans,"%d %s %d %d %d",&idtrans,&date,&sold,
              &returned,&bought) != EOF)
  {
    /* if no match keep writing and reading the master file */
    while (idtrans > idmast)
    {
      fprintf(newmast,"%d %d\n",idmast, balance);
      fscanf(oldmast,"%d %d", &idmast, &balance);
    }
    balance = balance + bought - sold + returned;
  }
  /* to get here the last transaction record has just been read */
  /* write the last updated new master file */
  fprintf(newmast,"%d %d\n",idmast, balance);
  /* write any remaining old master records to the new master */
  while (fscanf(oldmast,"%d %d", &idmast, &balance) != EOF)
  {
    fprintf(newmast,"%d %d",idmast, balance);
    fscanf(oldmast,"%d %d", &idmast, &balance);
  }
  fclose(oldmast);
  fclose(newmast);
  fclose(trans);
  printf("\n....File update complete...");
}
```

Exercises 8.3

1. Write a C program to create the pollen file illustrated in Figure 8–1.

2. An alternate update algorithm to the one used in Program 8-8 is to use two files: the first file is the pollen file illustrated in Figure 8–1 and the second file is the updated file. Using two files the update algorithm becomes:

```
open both files
read and add the first pollen
for the next nine pollen records
   read a pollen count
   add the count to the total
   write the count to the new file
endfor
request the current pollen count reading
write the current pollen count to the new file
close both files
calculate and display the previous ten day average
calculate and display the current ten day average.
```

Write a C file update program using this algorithm.

3 a. A file named `polar.dat` contains the polar coordinates needed in a graphics program. Currently this file contains the following data:

DISTANCE (INCHES)	ANGLE (DEGREES)
2	45
6	30
10	45
4	60
12	55
8	15

Write a C program to create this file on your computer system.

b. Using the `polar.dat` file created in Exercise 4a, write a C program that accepts distance and angle data from the user and adds the data to the end of the file.

c. Using the `polar.dat` file created in Exercise 3a, write a C program that reads this file and creates a second file named `xycord.dat`. The entries in the new file should contain the rectangular coordinates corresponding to the polar coordinates in the `polar.dat` file. Polar coordinates are converted to rectangular coordinates using the equations

$$x = r\cos(\theta)$$
$$y = r\sin(\theta)$$

where r is the distance coordinate and θ is the radian equivalent of the angle coordinate in the `polar.dat` file.

4 a. Write a C program to create the both the `oldbook.mas` file, illustrated in Table 8–4, and the `book.trn` file, illustrated in Table 8–5. (*Note:* Do not include the column headings in the file.)

b. Using the files created in Exercise 4a, enter and run Program 8-9 to verify its operation.

c. Modify Program 8-9 to prompt the user for the names of the old master file, the new master file, and the transaction file. The modified program should accept these file names as input while the program is executing.

d. Using the `book.trn` file create in Exercise 4a, write a C program that reads this file and displays the transaction data in it, including the heading lines shown in Table 8-5.

5. Modify Program 8-9 to use a single master file. Thus, as each record is updated it should be written back to the old master file. Any master file record that does not have to be updated should be left as it currently exists on the master file.

6 a. Write a C program to create a data file containing the following information:

Student ID Number	Student Name	Course Code	Course Credits	Course Grade
2333021	BOKOW, R.	NS201	3	A
2333021	BOKOW, R.	MG342	3	A
2333021	BOKOW, R.	FA302	1	A
2574063	FALLIN, D.	MK106	3	C
2574063	FALLIN, D.	MA208	3	B
2574063	FALLIN, D.	CM201	3	C
2574063	FALLIN, D.	CP101	2	B
2663628	KINGSLEY, M.	QA140	3	A
2663628	KINGSLEY, M.	CM245	3	B
2663628	KINGSLEY, M.	EQ521	3	A
2663628	KINGSLEY, M.	MK341	3	A
2663628	KINGSLEY, M.	CP101	2	B

b. Using the file created in Exercise 6a, write a C program that creates student grade reports. The grade report for each student should contain the student's name and identification number, a list of courses taken, the credits and grade for each course, and a semester grade point average. For example, the grade report for the first student is:

```
Student name: BOKOW, R.
Student ID Number: 2333021

Course      Course      Course
 Code       Credits     Grade
NS201          3          A
MG342          3          A
FA302          1          A

Total Semester Course Credits Completed: 7
Semester Grade Point Average: 4.0
```

The semester grade point average is computed in two steps. First, each course grade is assigned a numerical value (A = 4, B = 3, C = 2, D = 1, F = 0) and the sum of each course's grade value times the credits for each course is computed. This sum is then divided by the total number of credits taken during the semester.

7 a. Write a C program to create a data file containing the following information:

Student ID Number	Student Name	Course Credits	Grade Point Average (GPA)
2333021	BOKOW, R.	48	4.0
2574063	FALLIN, D.	12	1.8
2663628	KINGSLEY, M.	36	3.5

b. Using the file created in Exercise 7a as a master file and the file created in Exercise 6a as a transactions file, write a file update program to create an updated master file.

8.4 Text and Binary Files[6]

All of the files created in the previous three sections have been text files. The term *text file* refers to the type of codes used by the computer to store the data in the file. In addition to text files C permits the construction of *binary files*, which use a different set of codes for storing the data.

This section presents the specifics of text and binary data storage, describes the relative merits of each file type, and specifies the mechanics of designating a file's type when the file is opened.

Text File Storage

Each character in the files that we have been using is stored using a character code, such as the ASCII or EBCDIC codes introduced in Section 2.1. These codes assign a specific code to each letter in the alphabet, to each of the digits 0 through 9, and to special symbols such as the decimal point and dollar sign. The ASCII and EBCDIC uppercase letter codes were listed in Table 2–2. Table 8–6 lists the correspondence between the decimal digits 0 through 9 and their ASCII and EBCDIC representations in both binary and hexadecimal notation. Additionally, the ASCII and EBCDIC codes for a decimal point, blank space, carriage return, and line feed character are included in the table.

Using Table 8–6, we can determine how the decimal number 67432.83, for example, is stored in a data file using the ASCII code. In ASCII, this sequence of digits and decimal point requires eight character storage locations and is stored using the codes illustrated in Figure 8–4.

The advantage of using ASCII or EBCDIC codes for data files is that the file can be read and displayed by any word processing or editor program that is provided by your computer system. Such editor and word processing programs are called text editors because they are designed to process alphabetical text. The word processing program can read the ASCII or EBCDIC code in the data file and display the letter, symbol, or digit corresponding to the code. This permits a data file created in C to be examined and changed by other than C programs.

A text file is the default file type created in C when a file is opened. An option within the `fopen( )` function permits explicit selection of this file type, or selection of the alternative binary form. The explicit selection of a text file is made by adding the letter `t` after the mode (`rt`, `wt`, `at`, etc.) as the second argument in the `fopen( )` function call. As an example employing this option, assume that the following list of experimental results is to be stored in a text file named `exper.dat`:

Experiment Number	Result
1	8
2	12
3	497

[6] This topic may be omitted on first reading without loss of subject continuity.

TABLE 8–6 Selected ASCII Codes

Character	ASCII Binary Value	ASCII Hex. Value	EBCDIC Binary Value	EBCDIC Hex. Value
0	00110000	30	11110000	F0
1	00110001	31	11110001	F1
2	00110010	32	11110010	F2
3	00110011	33	11110011	F3
4	00110100	34	11110100	F4
5	00110101	35	11110101	F5
6	00110110	36	11110110	F6
7	00110111	37	11110111	F7
8	00111000	38	11111000	F8
9	00111001	39	11111001	F9
.	00101110	2E	01001011	4B
blank space	00100000	20	01000000	40
carriage return	00001101	0D	00001101	0D
line feed	00001010	0A	00001010	0A

```
36 37 34 33 32 2E 38 33
```

FIGURE 8–4 The Number 67432.83 Represented in ASCII CODE

Program 8-10 opens a file named `exper.dat` to store this data. Additionally, the `fopen( )` function is explicitly instructed to create a text file.

When Program 8-10 is executed, a file named `exper.dat` is created and saved by the computer. The file is a text file consisting of the following three lines:

```
1    8
2   12
3  497
```

The spacing between data items in the file is due to the formatting used in the `fprintf( )` function call.

The text file created by Program 8-10 contains 28 characters. These characters consist of the codes used to store the required digits (one code per digit or letter, which is the hallmark of a text file) plus the blank spaces before each number, a

Program 8-10

```
#include <stdio.h>
main( )
{
  int i, result[3] = {8, 12, 497};
  FILE *out_file;

  out_file = fopen("exper.dat","wt");
  for (i = 0; i < 3; ++i)
    fprintf(out_file,"%2d  %3d\n", i+1, result[i]);
  fclose(out_file);
}
```

carriage return and newline character at the end of each data line, and a special end-of-file (EOF) marker placed as the last item in the file when the file is closed.

Assuming characters are stored using the ASCII code listed in Table 8–6, the `exper.dat` data file is stored physically as shown in Figure 8–5. For convenience, the character corresponding to each hexadecimal code is listed below the code. Although the actual code used for the end-of-file (EOF) marker is system dependent, the hexadecimal codes `00` and `26` (Control-Z) are commonly used because they have no equivalent character representation.

Binary Files[7]

An alternative to text files, where each character in the file is represented by a unique code, are binary files. Binary files store numerical values using the computer's internal numerical code. For example, assume that the computer stores numbers internally using 16 bits in the two's complement format described in Section 1.7. Using this format the decimal number 8 is represented as the binary number `0000 0000 0000 1000`, the decimal number 12 as `0000 0000 0000 1100`, and the decimal number 497 as `0000 0001 1111 1011`.

The advantages of using this format are that no intermediary conversions are required for storing or retrieving the data and the resulting file usually requires less storage space than its text counterpart. The disadvantages are that the file can no longer be visually inspected using a text editing program or transferred between computers that use different internal number representations.

The explicit selection of a binary file is made by adding the letter `b` after the

FIGURE 8–5 The `exper.dat` File as Stored by the Computer

```
20 31 20 20 20 20 38 0D 0A 20 32 20 20 20 31 32 0D 0A 20 33 20 20 34 39 37 0D 0A 00
   1           8 CR LF    2          1  2 CR LF    3       4  9  7 CR LF EOF
```

[7] Before reading this section be sure that you are familiar with the computer storage concepts presented in Section 1.7.

```
     Header          1st Value       2nd Value       Trailer
| 00 00 00 10 | 00 00 00 01 | 00 00 00 08 | 00 00 00 10 | <- 1st Record
| 00 00 00 10 | 00 00 00 02 | 00 00 00 0C | 00 00 00 10 | <- 2nd Record
| 00 00 00 10 | 00 00 00 03 | 00 00 01 F1 | 00 00 00 10 | <- 3rd Record
| 00 00 00 00 | <- End of File Marker Record
```

FIGURE 8–6 The `exper.bin` File as Stored by the Computer

mode (`rb`, `wb`, `ab`, etc.) as the second argument in the `fopen( )` function call.[8] For example, the function call

```
fopen("exper.bin", "wb")
```

opens the file named `exper.bin` as an binary file.

If the data previously stored in the text file by Program 8-10 were written to a binary file, the file structure illustrated in Figure 8–6 would be produced. In this figure hexadecimal values are used to indicate the actual binary values that are stored. Although the figure separates the file's records into individual lines, with bars (|) used to distinguish individual items in each record, in actuality the file is stored as a consecutive sequence of codes.

As shown in the figure, each record in a binary file is preceded by a header value and followed by a trailer value. The values in the header and trailer are always equal. They contain the number of bytes in the record (each hexadecimal value is one byte in length—review Section 1.7 for a description of a byte). For example, the first record contains 16 bytes, which is indicated by the hexadecimal value `10`. Between each header and trailer value are the record's data items. As indicated in Figure 8–6, each record contains two integer values, with each integer stored using four bytes (32 bits). The hexadecimal values shown on the first line correspond to the decimal numbers 1 and 8, the values on the second line to the decimal numbers 2 and 12, and the values on the third line to the decimal numbers 3 and 497. These are the same values previously illustrated in Figure 8–5 using the ASCII code. Although the number of bytes used to store an integer is system dependent, the layout of all binary files corresponds to the form shown in Figure 8–6.

The fact that a file uses binary storage codes does not preclude its contents being displayed in text form. The file is simply read in its binary form and then displayed using `printf( )` function calls.

Exercises 8.4

1. Write individual `fopen( )` function calls to explicitly open files having the following characteristics:

a. A text file named `test.dat` that is to be assigned the internal file name `in_file`. The file is to be opened for reading.

b. A text file named `descri` that is to be assigned the internal file name `descrip`. The file is to be opened for writing.

c. A text file named `names` that is to be assigned the internal file name `out_file`. The file is to be opened for appending.

[8] On some systems binary access is not supported, in which case the file reverts to a text file.

d. A binary file named `types` that is to be assigned the internal file name `disktype`. The file is to be opened for reading.
e. A binary file named `codes` that is to be assigned the internal file name `idcodes`. The file is to be opened for writing.
f. A binary file named `balance.dat` that is to be assigned the internal file name `balances`. The file is to be opened for appending.

2. Redo Exercise 1 but omit all explicit file type designations from the `fopen( )` function call when the desired file type is correctly selected by C's default values.

3. Write, compile, and run a C program that writes the four floating point numbers 92.65, 88.72, 77.46, and 82.93 to a text file named `result`. After writing the data to the file your program should read the data from the file, determine the average of the four numbers read, and display the average. Verify the output produced by your program by manually calculating the average of the four input numbers.

4. If your system supports binary files, redo Exercise 3 using a binary file.

5 *a.* Write, compile, and execute a C program that creates a text file named `points` and writes the following numbers to the file:

```
 6.3   8.2   18.25   24.32 <— 1st record
 4.0   4.0   10.0    -5.0  <— 2nd record
-2.0   5.0    4.0     5.0  <— 3rd record
```

b. Using the data in the `points` file created in Exercise 5a write, compile, and run a C program that reads each record and interprets the first and second numbers in each record as the coordinates of one point and the third and fourth numbers as the coordinates of a second point. Using the formulas given in Exercises 25 and 26 of Section 2.3, have your program compute and display the slope and midpoint of the two points entered. Your program should use a `while` statement that uses the `EOF` marker as a sentinel.

6. If your system supports binary files, redo Exercise 5 using a binary file.

7 *a.* Write, compile, and run a C program that creates a text file named `grades` and writes the following numbers to the file:

```
100, 100, 100, 100
100, 0, 100, 0
86, 83, 89, 94
78, 59, 77, 85
89, 92, 81, 88
```

b. Using the data in the `grades` file created in Exercise 7a write, compile, and run a C program that reads each line in the `grades` file, computes the average for each line, and displays the average.

8. If your system supports binary files, redo Exercise 7 using a binary file.

8.5 Common Programming Errors

Four programming errors are common when using files. An extremely common error is to use the file's external name in place of the internal file name when accessing the file. The only standard library function that uses the data file's external name is the `fopen( )` function. All the other standard functions pre-

sented in this chapter require the variable name assigned to the file when it was initially opened.

The next error is to omit the file name altogether. Programmers used to the `scanf( )` and `printf( )` functions that access the standard input and output devices, where a file name is not required, sometimes forget to include a file name when accessing data files.

A third error occurs when using the `EOF` marker to detect the end of a file. Any variable used to accept the `EOF` must be declared as an integer variable, not a character variable. For example, if `ch` has been declared as a character variable the expression

```
while ( (c = getc(in_file)) != EOF)
```

produces an infinite loop. This occurs because a character variable can never take on an `EOF` code. `EOF` is an integer value that has no character representation. This ensures that the `EOF` code can never be confused with any legitimate character encountered as normal data in the file. To terminate the above expression, the variable `ch` must be declared as an integer variable.

The last error concerns the offset argument sent to the function `fseek( )`. This offset must be a long integer constant or variable. Any other value passed to `fseek( )` can result in an unpredictable effect.

8.6 Chapter Summary

1. A *data file* is any collection of data stored together in an external storage medium under a common name.
2. Data files are opened using the `fopen( )` library function. This function connects a file's external name with an internal file name. After the file is opened, all subsequent accesses to the file require the internal file name.
3. A file can be opened for reading, writing, or appending. A file opened for writing creates a new file or erases any existing file having the same name as the opened file. A file opened for appending makes an existing file available for data to be added to the end of the file. If the file does not exist it is created. A file opened for reading makes an existing file's data available for input.
4. An internal file name must be declared as a `FILE`. This means that a declaration of the type

   ```
   FILE  *file-name;
   ```

   must be included with the declarations in which the file is opened. `file-name` can be replaced with any user-selected variable name. The header file `stdio.h` contains the definition for File and must be included within a program that uses files.
5. In addition to any files opened within a function, the standard files `stdin`, `stdout`, and `stderr` are automatically opened when a program is run. `stdin` is the name of the physical file used for data entry by `scanf( )`, `stdout` is the name of the physical file device used for data display by

TABLE 8–7 Standard File Library Functions

Function Name	Purpose
`fopen( )`	Open or create a file
`fclose( )`	Close a file
`fgetc( )` and `getc( )`	Character input
`getchar( )`	Character input from `stdin`
`fgets( )`	String input
`gets( )`	String input from `stdin`
`fscanf( )`	Formatted input
`scanf( )`	Formatted input from `stdin`
`fputc( )` and `putc( )`	Character output
`putchar( )`	Character output to `stdout`
`fputs( )`	String output
`puts( )`	String output to `stdout`
`fprintf( )`	Formatted output
`printf( )`	Formatted output to `stdout`
`fseek( )`	File positioning
`rewind( )`	File positioning
`ftell( )`	Position reporting

`printf( )`, and `stderr` is the name of the physical file device used for displaying system error messages.

6. Data files can be accessed randomly using the `rewind( )`, `fseek( )`, and `ftell( )` functions.
7. Table 8–7 lists the standard file library functions.

8.7 Enrichment Study: Control Codes

In addition to responding to the codes for letters, digits, and special punctuation symbols, which are collectively referred to as *printable characters*, physical device files such as printers and CRT screens can also respond to a small set of control codes. These codes, which convey control information to the physical device,

have no equivalent characters that can be displayed, thus they are called *non-printable characters.*

Two of these codes, which are extremely useful in applications, are the form-feed and bell control codes. When the form-feed control code is sent to a printer, the printer ejects a page of paper and begins printing on the next sheet of paper. If you take care to align the printer to the top of a new page when printing begins, the form-feed control character can be used as an equivalent "top-of-page" command. When the equivalent clear code is sent to a CRT display, the screen is cleared of all text and the cursor is positioned at the left-hand corner of the screen.

Sending control codes to an output device is done in a manner similar to sending a printable character to a file. Recall that sending a printable character to a file requires two pieces of information: the file name and the character being written to the file. For example, the statement `fputc('a',out_file);` causes the letter `a` to be written to the file named `out_file`. Instead of including the actual letter as an argument to `fputc( )`, we can substitute the numerical storage code for the letter. For computers that use the ASCII code, this amounts to substituting the equivalent ASCII numerical value for the appropriate letter. Referring to Appendix F, we see that in the ASCII code the value for `a` is 97 as a decimal number, 61 as a hexadecimal number, and 141 as an octal number. Any one of these numerical values can be used in place of the letter `a` in the previous `fputc( )` function call. Thus, the following four statements are all equivalent:

```
fputc('a',out_file);
fputc(97, out_file);
fputc(0x61 out_file);
fputc('\141',out_file);
```

Note that in each of these statements we have adhered to the notation used by C in identifying decimal and hexadecimal numbers. A number with no leading `0` is considered a decimal number and a number with a leading `0x` is considered a hexadecimal value. Octal character codes, however, must be preceded by a backslash and enclosed in single apostrophes.

The importance of substituting the numerical code for the letter is only realized when a control code rather than a character code must be sent. Since no equivalent character exists for control codes, the actual code for the command must be used. Although each computer can have its own code for clearing the CRT screen, the bell code and the printer form-feed code are fairly universal. To activate the bell, the octal code 07 is used. The octal form-feed code for most printers is 014. Thus, if the file `out_file` has been opened as the printer in write mode, the statement

```
fputc('\014',out_file);
```

causes the printer to eject the current page. Similarly, if `scrn` has been opened as the CRT screen in write mode, the statement

```
fputc('\07',scrn);
```

causes the bell to be activated for a short "beep."

For personal computers and compatible machines, the CRT screen has its own clear code. For your computer, check the manual for the CRT screen to obtain the proper clear-screen control code. You must also check the name by which your computer "knows" the printer and CRT screen. For IBM personal computers the printer has the name `prn` and the CRT screen the name `con` (short for console). Program 8-11 illustrates the use of control codes to eject a page of paper from the printer and alert the user with a "beep" if the printer is not turned on. Using `#define` commands, the appropriate codes have been equated to more readable symbolic names.

Program 8-11

```
#include <stdio.h>
#define BELL '\07'
#define TOPOFPAGE '\014' /* page eject code */
main( )
{
  FILE *printer;

  printer = fopen("prn", "w");

  if(printer == 0)          /* check that the file has been opened */
  {
    fputc(BELL,stdout);
    printf("The printer cannot be opened for output.");
    printf("\nPlease check the printer is on and ready for use.");
  }
  else
    fputc(TOPOFPAGE,printer);
}
```

The `if-else` statement in Program 8-11 is used to ensure that the printer has been opened and is ready for output. The symbolic constants `BELL` and `TOPOF-PAGE` can be used freely within the program because they have been properly defined (see Section 3.5). Since the CRT screen is the standard output device for the computer used to run Program 8-11, the CRT did not have to be opened as a new file. Instead, the file name `stdout` was used to send the `BELL` constant to the screen.

In addition to the Bell code, all CRT screens have control codes to position the cursor directly at different screen locations. This enables the programmer to place messages anywhere on the screen. Since these codes differ for various CRT models, you should check the manual for your computer to determine the proper codes. Additionally, many C compilers for personal computers include standard library functions that provide the same cursor-positioning capabilities.

Numerical Techniques and Applications

Chapter Nine

9.1 Solving Simultaneous Linear Equations
9.2 Root Finding
9.3 Numerical Integration
9.4 Common Programming Errors
9.5 Chapter Summary

In this chapter we present and apply several programming techniques to solve a variety of commonly encountered numerical applications. First we look at how to solve simultaneous equations; next, we turn to root finding and numerical integration techniques.

9.1 Solving Simultaneous Linear Equations

A *linear equation in two unknowns*, x and y, is an equation of the form

$$ax + by = k \tag{1}$$

where a, b, and k are known numbers and a and b are not both zero. For example, the equation $3x + 2y = 10$ is linear because it has the form of Equation 1 with $a = 3$, $b = 2$, and $k = 10$. Although we have defined a linear equation in two unknowns, the definition may be extended to include any number of unknowns. For example, the equation $x + 3y + 2z = 5$ is a linear equation in the three unknowns x, y, and z. What makes the equation linear is that each unknown quantity is raised only to the first power and multiplied by a known number.

Two Linear Equations with Two Unknowns

A simultaneous set of two linear equations in two unknowns are two linear equations having the general form

$$\begin{aligned} a_1x + b_1y &= k_1 \\ a_2x + b_2y &= k_2 \end{aligned} \tag{2}$$

where x and y are the unknowns. For example, the equations

$$\begin{aligned} 2.3x + \ \ 4y &= 21.75 \\ 3x + 1.5y &= 13.5 \end{aligned}$$

are a simultaneous set of two linear equations in two unknowns. Here the constants a_1, b_1, k_1, a_2, b_2, k_2 are the numbers 2.3, 4, 21.75, 3, 1.5, and 13.50, respectively, and the unknowns are the values of x and y that satisfy both equations.

The solution to two simultaneous linear equations in two unknowns, if one exists, can easily be solved for using *Cramer's Rule*, which requires the use of determinants. A *determinant* is a square array of elements enclosed in straight lines, such as:

$$\begin{vmatrix} 3 & 6 \\ 2 & 5 \end{vmatrix}$$

that has the same number of rows as it has columns and can be evaluated to yield a result. The number of rows or columns in a determinant determines its *order*. For example, the determinant

$$\begin{vmatrix} 6 & 9 \\ 0 & -2 \end{vmatrix}$$

is a second-order determinant, because it has two rows and two columns. The general form of a second-order determinant is:

$$\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}$$

where a_1 and b_1 refer to the first and second elements in the first row, respectively, and a_2 and b_2 refer to the first and second elements of the second row, respectively. The value of this determinant is then calculated as $a_1 \cdot b_2 - a_2 \cdot b_1$. Thus, the value of the determinant

$$\begin{vmatrix} 3 & 6 \\ 2 & 5 \end{vmatrix}$$

is $(3 \cdot 5) - (2 \cdot 6) = 3$, and

$$\begin{vmatrix} 6 & 9 \\ 0 & -2 \end{vmatrix} = (6 \cdot -2) - (0 \cdot 9) = -12$$

Let us now relate the evaluation of determinants to solving sets of two linear equations. Cramer's rule states that the solution of the set of linear equations

$$a_1 x + b_1 y = k_1$$
$$a_2 x + b_2 y = k_2$$

is:

$$x = \frac{\begin{vmatrix} k_1 & b_1 \\ k_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}$$

and

$$y = \frac{\begin{vmatrix} a_1 & k_1 \\ a_2 & k_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}$$

As an example using these formulas, consider the set of linear equations

$$2x + 3y = 130$$
$$10x + 5y = 330$$

Using Cramer's rule we obtain the solution for this set of equations as:

$$x = \frac{\begin{vmatrix} 130 & 5 \\ 330 & 5 \end{vmatrix}}{\begin{vmatrix} 2 & 3 \\ 10 & 5 \end{vmatrix}} = \frac{(130 \cdot 5) - (330 \cdot 3)}{(2 \cdot 5) - (10 \cdot 3)} = \frac{650 - 990}{10 - 30} = \frac{-340}{-20} = 17$$

$$y = \frac{\begin{vmatrix} 2 & 130 \\ 10 & 330 \end{vmatrix}}{\begin{vmatrix} 2 & 3 \\ 10 & 5 \end{vmatrix}} = \frac{(2 \cdot 330) - 100 \cdot 130)}{(2 \cdot 5) - (10 \cdot 3)} = \frac{660 - 1300}{10 - 30} = \frac{-640}{-20} = 32$$

Notice that the same determinant is used as the denominator in solving for both unknowns. When this denominator determinant is equal to zero, no unique solution can be found that solves the equations. Also notice that the numerator used in solving for x is the same as the denominator, with the first column replaced by the coefficients on the right side of the equations (k_1 and k_2). Likewise, the numerator in the solution for y replaces the second column of the denominator determinant with the coefficients k_1 and k_2. This pattern lends itself to ease of programming for the solution of equations in three variables.[1]

Program 9-1 allows us to enter the numerical coefficients for two linear equations and solve for the two unknowns, x and y, when a unique solution exists. The values of all three determinants required to solve the set of two linear equations are evaluated using a single C function named `det2( )`.

Following is the output produced by Program 9-1 when it is used to solve this set of equations:

$$2x + 3y = 130$$
$$10x + 5y = 330$$

```
Enter a1, b1, k1, a2, b2, k2: 2 3 130 10 5 330

The solution is:
x = 17.000000
y = 32.000000
```

Application: Battery Charger

As an example of solving two equations with two unknowns consider the electrical equivalent circuit of an automobile battery charger shown in Figure 9–1. The circuit models the car's electrical system after the car has been started. The purpose of the generator (or alternator) is to supply power to (i.e., recharge) the battery. A real voltage source, such as a generator or battery, can be modeled by an

[1] For larger systems of equations Cramer's rule is inefficient and Gaussian elimination is the preferred method of solution.

Program 9-1

```
#define EPSILON 0.001
#include <stdio.h>
#include <math.h>
main( )
{
  float det2(float, float, float, float);  /* prototype */
  float a1, b1, k1, a2, b2, k2, x, y, valdet;

  printf("Enter a1, b1, k1, a2, b2, k2: ");
  scanf("%f %f %f %f %f %f", &a1,&b1,&k1,&a2,&b2,&k2);
  valdet = det2(a1,b1,a2,b2);
  if ( fabs(valdet) >= EPSILON)
  {
    x = det2(k1,b1,k2,b2) / valdet;
    y = det2(a1,k1,a2,k2) / valdet;
    printf("\nThe solution is:\n");
    printf("x = %f\n", x);
    printf("y = %f\n", y);
  }
  else
    printf("\nA unique solution does not exist.\n");
}
float det2(float a, float b, float c, float d)
/* function to evaluate the 2 x 2 determinant    | a  b | */
/*                                               | c  d | */
{
  float valdet2;

  valdet2 = a * d - b * c;
  return(valdet2);
}
```

ideal source in series with a resistor representing the internal resistance of that source. The components shown in the circuit diagram are listed below:

V_G—ideal generator voltage
R_G—generator internal resistance
V_B—ideal battery voltage (e.g., 12 volts)
R_B—battery internal resistance
R_L—load resistance (i.e., equivalent resistance of all devices drawing energy, such as headlights and radio)

If the component values are known, we can set up two linear equations in terms of the two loop currents, i_1 and i_2. By solving for the two unknowns we can

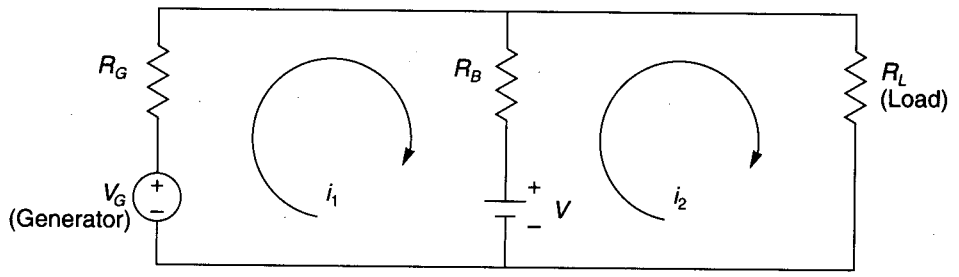


FIGURE 9–1 Automobile Battery Charging Circuit

determine the voltage, current, or power for any source or resistor. By applying circuit laws to the two loops we obtain the following equations:

$$-V_G + R_G i_1 + R_B(i_1 - 1_2) + V_B = 0$$
$$-V_B + R_B\,(i_2 - i_1) + R_L i_2 = 0$$

These equations result from the fact that the sum of all voltages around a closed circuit path (i.e., a loop) is zero, and that the voltage across a resistor (in volts) is equal to its current (in amperes, or amps for short) multiplied by the resistance value (in ohms). We notice that the current through the generator is i_1 flowing upward and the load current is i_2 flowing downward. The battery has both loop currents circulating through it in opposite directions, so that its current can be specified as $i_1 - i_2$ flowing downward, or equivalently $i_2 - i_1$ flowing upward.

The equations can be easily rearranged to fit the standard form shown previously. This results in

$$(R_G + R_B)\,i_1 \quad - R_B\,i_2 = V_G - V_B$$
$$-R_B\,i_1 + (R_B + R_L)\,i_2 = V_B$$

In terms of previous notation we see that

$$a_1 = R_G + R_B$$
$$b_1 = a_2 = -R_B,$$
$$b_2 = R_2 + R_L$$
$$k_1 = V_G - V_B$$
$$k_2 = V_B$$

Suppose we want to determine whether the battery is charging or discharging its energy. If the battery is being charged by the generator its net current flows downward; that is the case if current i_1 is larger than i_2. Otherwise, if i_2 is larger than i_1, current flows upward and the battery is discharging. Once i_1 and i_2 have been solved for, a program segment such as

```
if(i1 > i2)
  printf("\nThe battery is charging");
```

```
        else
          printf("\nThe battery is discharging");
```

displays the battery's status.

Program 9-2 allows us to enter the voltage source and resistor values, calculates the two loop currents, and determines whether the battery is charging or discharging.

Program 9-2

```
#include <stdio.h>
main( )
{
  float det2(float, float, float, float); /* prototype */
  float a1, b1, k1, a2, b2, k2, valdet;
  float vg, vb, rg, rb, rl, i1, i2;

  printf("Enter vg and vb in volts: ");
  scanf("%f %f", &vg, &vb);
  printf("Enter rg, rb, and rl in ohms: ");
  scanf("%f %f %f", &rg, &rb, &rl);
  a1 = rg + rb;
  b1 = -rb;
  a2 = -rb;
  b2 = rb + rl;
  k1 = vg -vb;
  k2 = vb;
  valdet = det2(a1,b1,a2,b2);
  i1 = det2(k1,b1,k2,b2) / valdet;
  i2 = det2(a1,k1,a2,k2) / valdet;
  printf("\nGenerator Current = %f amps", i1);
  printf("\nLoad Current = %f amps",i2);

  if(i1 > i2)
    printf("\nThe battery is charging");
  else
    printf("\nThe battery is discharging");
}
float det2(float a, float b, float c, float d)
/* function to evaluate the 2 x 2 determinant | a  b | */
/*                                            | c  d | */
{
  float valdet2;

  valdet2 = a * d - b * c;
  return(valdet2);
}
```

The results of two sample runs are shown below, using realistic values for voltages and resistances.

```
Enter vg and vb in volts: 12.5 12
Enter rg, rb, and rl in ohms: 0.15 0.1 1

Generator Current = 6.603773 amps
Load Current = 11.509433 amps
The battery is discharging
```

and

```
Enter vg and vb in volts: 15 12
Enter rg, rb, and rl in ohms: 0.15 0.1 1

Generator Current = 16.981131 amps
Load Current = 12.452829 amps
The battery is charging
```

Further analysis of this problem shows that for a 12-volt car battery and the given resistor values, a generator voltage of at least 13.8 volts is required to keep the battery charging. The generator voltage must be higher than the battery voltage to overcome the loss in the internal source resistances.

Three Equations with Three Unknowns

Cramer's Rule also applies to the solution of three linear equations in three unknowns, having this general form:

$$\begin{aligned} a_1x + b_1y + c_1z &= k_1 \\ a_2x + b_2y + c_2z &= k_2 \\ a_3x + b_3y + c_3z &= k_3 \end{aligned}$$

As with the case of two linear equations in two unknowns, each unknown in a set of three linear equations in three unknowns can be evaluated as a ratio of two determinants. In this case, however, each determinant is of third order. Specifically, for the equations above, the solution is:

$$\begin{aligned} x &= det_x \,/\, det_3 \\ y &= det_y \,/\, det_3 \\ z &= det_z \,/\, det_3 \end{aligned}$$

where

$$det_x = \begin{vmatrix} k_1 & b_1 & c_1 \\ k_2 & b_2 & c_2 \\ k_3 & b_3 & c_3 \end{vmatrix}$$

$$det_y = \begin{vmatrix} a_1 & k_1 & c_1 \\ a_2 & k_2 & c_2 \\ a_3 & k_3 & c_3 \end{vmatrix}$$

$$det_z = \begin{vmatrix} a_1 & b_1 & k_1 \\ a_2 & b_2 & k_2 \\ a_3 & b_3 & k_3 \end{vmatrix}$$

$$det_3 = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$$

As we saw in the two-equation case, the numerator determinant corresponding to each unknown is evaluated by replacing the entries of the appropriate column of the denominator determinant (det_3) by the coefficients on the right side of the equations (k_1, k_2, and k_3). As in the two-equation case, when the denominator determinant is zero, no unique solution exists.

The actual evaluation of a third-order determinant can be made by multiplying each element of any arbitrarily chosen row or column by the remaining second-order determinant after removing the row and column corresponding to the element. The selected row or column is called the *pivot row* or *pivot column*, respectively, and the resulting evaluation proceeds by alternating additions and subtractions. For example, arbitrarily selecting the first column as a pivot column, the 3×3 determinant

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

can be evaluated as:

$$a \cdot \begin{vmatrix} e & f \\ h & i \end{vmatrix} - d \cdot \begin{vmatrix} b & c \\ h & i \end{vmatrix} + g \cdot \begin{vmatrix} b & c \\ e & f \end{vmatrix}$$

Program 9-3 uses this evaluation method to compute the determinants required to solve three linear equations in three unknowns. This permits the program to make use of the function `det2( )` previously developed for Program 9-1.

Following is a sample run of Program 9-3 used to solve this system of linear equations:

$$\begin{aligned} x + y + z &= 32.5 \\ 10x + 5y + 4z &= 240 \\ 1.5x + 0y \quad 10z &= 49 \end{aligned}$$

```
Enter a1, b1, c1, a2, b2, c2, a3, b3, c3
1 1 1 10 5 4 1.5 0 10
Enter k1, k2, and k3 32.5 240 49
The solution is:
x = 16.000000
y = 14.000000
z = 2.500000
```

Program 9-3

```
#include <stdio.h>
#include <math.h>
main( )
{
  float det3(float, float, float, float, float,
             float, float, float, float);
  float a1, b1, c1, a2, b2, c2, a3, b3, c3;
  float k1, k2, k3, x, y, z, valdet;

  printf("Enter a1, b1, c1, a2, b2, c2, a3, b3, c3\n");
  scanf("%f %f %f %f %f %f", &a1,&b1,&c1,&a2,&b2,&c2);
  scanf("%f %f %f",&a3,&b3,&c3);
  printf("Enter k1, k2, and k3\n");
  scanf("%f %f %f", &k1, &k2, &k3);
  valdet = det3(a1,b1,c1,a2,b2,c2,a3,b3,c3);
  if ( fabs(valdet) >= 0.001 )
  {
    x = det3(k1,b1,c1,k2,b2,c2,k3,b3,c3) / valdet;
    y = det3(a1,k1,c1,a2,k2,c2,a3,k3,c3) / valdet;
    z = det3(a1,b1,k1,a2,b2,k2,a3,b3,k3) / valdet;
    printf("The solution is:\n");
    printf("x = %f\n", x);
    printf("y = %f\n", y);
    printf("z = %f\n", z);
    }
  else
    printf("A unique solution does not exist.\n");
}
/*                                                          */
float det3(float a, float b,float c, float d, float e, float f,
           floatg,float h, float i)
/* function to evaluate the 3 x 3 determinant  | a  b  c | */
/*                                             | d  e  f | */
/*                                             | g  h  i | */
{
  float det2(float, float, float, float); /* prototype */
  float valdet3;

  valdet3 = a*det2(e,f,h,i)-d*det2(b,c,h,i)+g*det2(b,c,e,f);
  return(valdet3);
}
/*                                                          */
float det2(float a, float b, float c, float d)
/* function to evaluate the 2 x 2 determinant   | a  b | */
/*                                              | c  d | */
{
  float valdet2;

  valdet2 = a * d - b * c;
  return(valdet2);
}
```

Exercises 9.1

1. Compile and run Program 9–1 on a computer. Enter the data necessary to solve these linear equations:

$$3x + 4y = 5$$
$$3x + 5y = 4$$

2a. Notice that the two equations below cannot be solved:

$$x + 2y = 5$$
$$3x + 6y = 8$$

On the other hand, the equations

$$x + 2y = 5$$
$$3x + 6y = 15$$

result in many solutions. Can you see what the problem is?

b. Compile and run Program 9-1 for the equations in Exercise 2a.

3. Modify Program 9-1 to evaluate the second-order determinant using an array to pass the four coefficients rather than four individual scalars. Compile and run the modified program to verify the results produced by Program 9-1.

4. Compile and run Program 9-2 on a computer for a 12-volt battery charger with the following resistor values in ohms:

$$R_G = 0.2 \qquad R_B = 0.2 \qquad R_L = 1.0$$

Noting that 12 must be entered for V_B, enter a higher value of volts for V_G. Rerun the program, varying only the generator voltage. By trial and error find the minimum V_G required (to one decimal place) to cause the battery to charge rather than discharge.

5a. Modify Program 9-2 to allow you to enter values for V_B, R_G, R_B, and R_L from the keyboard. The currents i_1 and i_2 are to be calculated for all values of V_G ranging from 12.5 to 15.0 volts in steps of 0.1 volt. Rather than displaying a message as to whether the battery is charging or discharging, print the results in a table with the headings shown below:

```
   VG          I1          I2
(volts)     (amps)      (amps)
-------     ------      ------
```

b. Compile and run the modified program on a computer. Enter these values:

$$V_B = 12 \qquad R_G = 0.15 \qquad R_B = 0.1 \qquad R_L = 1$$

c. Repeat Exercise 5b for the values used in Exercise 5.

6. Compile and run Program 9-3 on a computer. Enter the data necessary to solve these linear equations:

$$2x + y - 2z = 8$$
$$x - 5y + 3z = 6$$
$$3x - 3y - 2z = 10$$

7. A common numerical problem is to find the equation of a straight line that best fits a set of n data points, denoted as (x_1,y_1), (x_2,y_2), (x_3,y_3), . . . (x_n,y_n). The equation of a straight line is given by $y = mx + b$, where m is the slope of the line and b is called the y-intercept. One technique for determining the values of m and b is called a *linear least-squares fit*. For such a fit the unknowns m and b are related by the set of two simultaneous linear equations:

$$nb + \left(\sum_{i=1}^{n} x_i\right)m = \sum_{i=1}^{n} y_i$$

$$\left(\sum_{i=1}^{n} x_i\right)b + \left(\sum_{i=1}^{n} x_i^2\right)m = \sum_{i=1}^{n} x_i y_i$$

a. Using Program 9-1 as a starting point, write a C program that accepts the given x and y values as inputs, determines the coefficients of the two equations, and then solves for the values of m and b. (*Hint:* Accept the number of data points as the first input and store the actual data points in two arrays named `x` and `y`, respectively.) Test your program using these data points: (1,0.5), (2,1.5), (3,1), (4,2).

b. Using the program developed for Exercise 7a, determine the equation of the straight line that best fits the following data points: (1,3), (2,1), (3,2), (4,1), (6,2), (8,5).

8. Experimental results on an unknown resistor produced the following table of voltages and currents:

Voltage (volts)	Current (amps)
1	0.018
2	0.043
3	0.056
4	0.085
5	0.092
6	0.100
7	0.102

The equation relating voltage, V; current, I; and resistance, R, is given by Ohm's Law, which states that $V = R \cdot I$. Use the program developed in Exercise 7a to assist you in finding the "best" guess at the resistance value, R.

9. Fitting a quadratic curve to a set of n data points, denoted as (x_1,y_1), (x_2,y_2), (x_3,y_3), . . . (x_n,y_n), requires determining the values of a, b, and c for the equation $y = a + bx + cx^2$ that fits the data in some best manner. One technique for determining the values of a, b, and c is called the *quadratic least-squares fit*. For such a fit, the unknowns a, b, and c are related by the set of equations:

$$na + b\sum_{i=1}^{n} x_1 + c\sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} y_i$$

$$a\sum_{i=1}^{n} x_i + b\sum_{i=1}^{n} x_i^2 + c\sum_{i=1}^{n} x_i^3 = \sum_{i=1}^{n} x_i y_i$$

$$a\sum_{i=1}^{n} x_i^2 + b\sum_{i=1}^{n} x_i^3 + c\sum_{i=1}^{n} x_i^4 = \sum_{i=1}^{n} x_i^2 y_i$$

Using Program 9-3 as a starting point, write a C program that accepts the given x and y values as inputs, determines the coefficients of the three equations, and then solves for the values of a, b, and c. (*Hint:* Accept the number of data points as the first input and store the actual data points in two arrays named `X` and `Y`, respectively.) Test your program using these data points: (1,0.5), (2,1.5), (3,1), (4,2).

10a. The official United States population for each census taken from 1900 to 1980 is listed below. The population figures are in millions of people and are rounded off to one fractional digit.

Year	U.S. Population
1900	76.2
1910	92.2
1920	106.0
1930	123.2
1940	132.2
1950	151.3
1960	179.3
1970	203.3
1980	226.5

Using the program developed for Exercise 9, determine the equation of the least-squares quadratic curve for this data.

b. Using the equation determined in Exercise 10a, determine an estimate for the population of the United States in the year 2000.

9.2 Root Finding

Although the solution to sets of linear equations can be obtained using method's such as Cramer's Rule presented in Section 9.1, and individual quadratic equations, such as

$$x^2 + 2x - 35 = 0$$

can be solved for x with the aid of the quadratic formula (see Section 4.5), no such computationally simple solutions exist for equations such as

$$x^4 + 5x^3 + 12x^2 - 7x + 21 = 0$$

and

$$\sin(5x) - 3x^2 + e^{4.8x} = 12$$

Although both of these equations have only a single unknown quantity, they are extremely difficult to solve because they both are nonlinear equations. By *solve*, of course, we mean finding a value for x that when substituted into the equation yields a value for the equation's left-hand side equal to that on its right-hand side. More formally, such a value of x is called a *root* of the equation. Not unexpectedly, then, the methods of solving these equations are referred to as *root-finding methods*.

All root-finding methods require that the equation whose roots are being sought be written as the equation of a curve. In general, this involves the following two steps:

Step 1: Rewrite the equation to have all of the unknowns and constants on one side of the equation and a zero on the other side.
For example, the equation $x^2 + 2x = 35$ can be rearranged in this form by subtracting 35 from both sides of the equation to yield $x^2 + 2x - 35 = 0$.
Step 2: Set the side of the equation with the unknowns equal to another variable, such as y.
For example, setting $y = x^2 + 2x - 35$ converts the original equation into the equation of a curve. For each value of x in this equation a value of y can be computed.

Once the given equation has been transformed into the equation of a curve, the root-finding problem reduces to locating values of x for which y is zero. Two such methods are now presented.

Both of these methods, as indeed all root-finding methods, rely on guessing at a value of x, finding the corresponding value of y, and then modifying the value of x until a y value of zero is reached. The difference in methods is based on the procedure used to modify each guess until a root is located.

Fixed Increment Iterations

In the fixed increment root-finding method each value of x is obtained from the previous value by adding or subtracting a fixed amount. To understand how this procedure works, consider Program 9-4, which tabulates values of the curve $y = x^2 + 2x - 35$ for integer values of x ranging from -10 to $+10$.

Program 9-4

```
#include <stdio.h>
#include <math.h>
main( )
{
  int x, y;

  printf("X VALUE       Y VALUE\n");
  printf("-------       -------\n");
  for(x = -10; x <= 10; ++x)
  {
    y = pow(x,2) + 2 * x - 35;
    printf(" %4d          %4d\n", x, y);
  }
}
```

The output produced by Program 9-4 is:

```
X VALUE      Y VALUE
-------      -------
  -10           45
   -9           28
   -8           13
   -7            0
   -6          -11
   -5          -20
   -4          -27
   -3          -32
   -2          -35
   -1          -36
    0          -35
    1          -32
    2          -27
    3          -20
    4          -11
    5            0
    6           13
    7           28
    8           45
    9           64
   10           85
```

The roots of the curve correspond to these values of x for which the calculated values of y are zero, in this case $x = -7$ and $x = 5$. As shown in Figure 9–2, which is a plot of the curve using the tabulated values, the roots are the values of x corresponding to the points where the curve intersects the x-axis, namely at $x = -7$ and at $x = 5$.

Now let us modify Program 9-4 to display the two roots without listing the entire table. We do this by eliminating the printing of the headings in Program 9-4 and replacing the `printf( )` function call inside the `for` loop with a conditional `printf( )` call that displays a message only if `y` is zero. Program 9-5 accomplishes this.

The output produced by this program is the following:

```
A root is at x = -7
A root is at x = 5
```

Of course it should be apparent that Program 9-5 found the solutions to the equation for us because the roots happen to be integers. Noninteger roots cannot be located by this routine because x is declared as an integer, so that only integer values are entered into the equation. To consider the more general case of noninteger roots, we consider the equation

$$x^2 + 2.1x - 16.96 = 0$$

From the quadratic formula the roots of this equation are $x = 5.3$ and $x = 3.2$. However, let us assume that we don't know where the roots are, as would be the

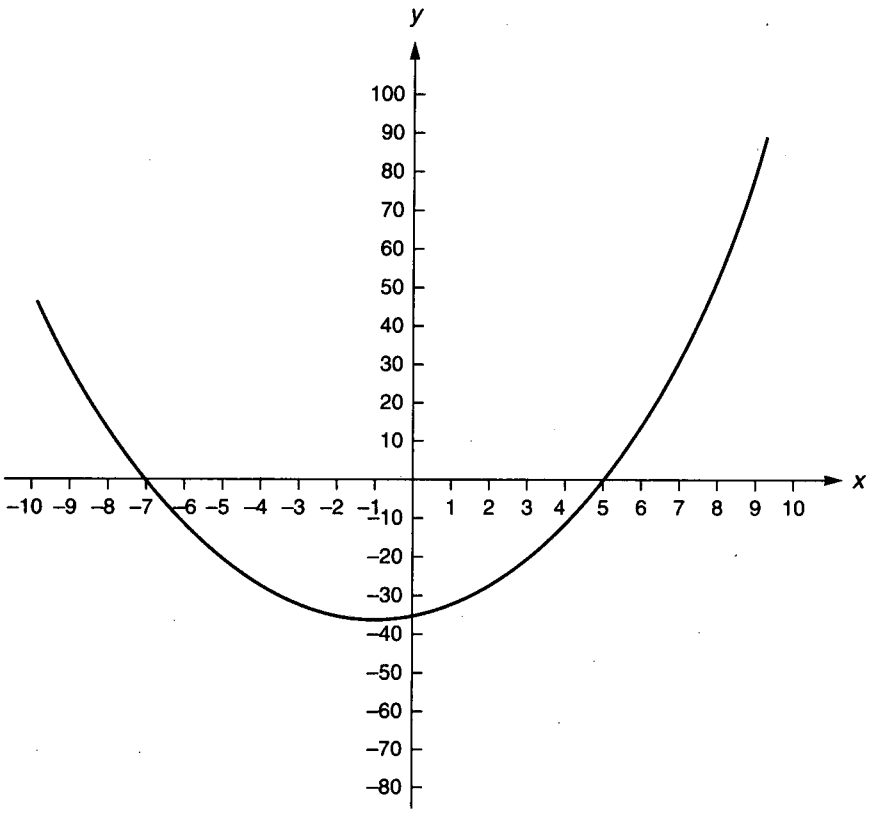


FIGURE 9–2 The Graph of the Curve $y = x^2 + 2x - 35$

Program 9-5

```
#include <stdio.h>
#include <math.h>
main( )
{
  int x,y;

  for (x = -10; x <= 10; ++x)
  {
    y = pow(x,2.0) + 2 * x - 35;
    if ( y == 0 ) printf("A root is at x = %d\n", x);
  }
}
```

case for a higher order equation or one with trigonometric or exponential functions. For this more general noninteger case, the variables `x` and `y` in the program need to be declared as float variables. The step size in the `for` loop, which is one

in Program 9-5, needs to be decreased to enable the finding of noninteger root locations. Also, we need to decide how close to zero the function needs to be to qualify a value as a root (recall from Section 4.1 that due to precision errors exact equality to zero is not always possible for real values). In Program 9-6 the step size used in the loop (called `incr`) and the allowable deviation from zero, referred to as the *allowable error*, are read from the keyboard. The value of `y` is calculated within the loop and the root value is printed out only if the absolute value of `y` does not exceed the allowable error.

Program 9-6

```
#include <stdio.h>
#include <math.h>
main( )
{
  float x, y, incr, error;

  printf("Enter the step size: ");
  scanf("%f", &incr);
  printf("Enter the allowable error: ");
  scanf("%f", &error);
  for ( x = -10; x <= 10; x += incr)
  {
    y = pow(x,2.0) + 2.1 * x - 16.96;
    if (fabs(y) <= error) printf("A root is at x = %f\n", x);
  }
}
```

The results of three sample runs using Program 9-6 are:

```
Enter the step size: 0.01
Enter the allowable error: 0.1
A root is at x = -5.309893
A root is at x = -5.299892
A root is at x = -5.289892
A root is at x = 3.190131
A root is at x = 3.200131
A root is at x = 3.210131
```

and

```
Enter the step size: 0.01
Enter the allowable error: 0.01
A root is at x = -5.299892
A root is at x = 3.200131
```

and

```
Enter the step size: 0.01
Enter the allowable error: 0.001
A root is at x = -5.299892
```

The results illustrate the care that needs to be taken in choosing both the step size and the allowable error. The step size determines the total number of iterations that are made; the allowable error determines the range of y values that are close enough to zero to qualify a given x value as a root. As shown in the first run, too large an allowable error may qualify too many roots, while as shown in the last run, too small a value may cause valid roots to be missed. In this case the root location at 3.2, due to roundoff error, is not found.

Although the second run correctly locates the two roots of the equation using a step size of `0.01` and an allowable error of the same value, this relationship between step size and allowable error cannot be generalized. However, the problem of selecting too large or too small a value for the allowable function error can be eliminated by using a different approach.

Referring to Figure 9–2, notice that the crossing of the x-axis by the curve causes the sign of y to change. The table of values printed by Program 9-4 also shows the sign changes. As x is increased from -10 the values of y are positive, and then change to negative as the root $x = -7$ is passed. The values of y remain negative until the second root $x = 5$ is reached, whereupon y is again positive for larger values of x. All curves exhibit sign changes in y when a root is encountered, with one exception. The exception is the occurrence of a curve that is tangent to the x-axis at some point, whereupon the sign of y will not change even though a root exists (this is referred to as a repeated root).

Taking advantage of the sign change feature allows us to have our program calculate two successive values of y within the loop, corresponding to a value of x and to x plus a chosen increment. If the two values of y differ in sign, a root is identified. The root, however, is between the two values of x used. Rather than simply use one of the values we obtain a more accurate computation of the root using interpolation. In Figure 9–3 a small portion of a curve is shown for both a positive slope and negative slope case; x_1 and x_2 represent two successive values of x, between which a root exists. The actual root location x_r is approximated by drawing a straight line between the points (x_1, y_1) and (x_2, y_2). Where this interpolation line crosses the x-axis determines x_r. From the triangles drawn in Figure 9–3 the slopes of the lines can be equated, resulting in the equation

$$\frac{y_1}{x_r - x_1} = \frac{-y_2}{x_2 - x_r}$$

for the negatively sloped case (the equation is the same for the positively sloped case because the minus sign is simply transposed from y_2 to y_1).

By solving the equation above for x_r the approximate root value is:

$$x_r = \frac{x_2 y_1 - x_1 y_2}{y_1 - y_2} = x_1 + \frac{y_1}{y_1 - y_2}(x_2 - x_1)$$

Program 9-7 incorporates this interpolation formula to locate the roots of the curve $y = x^2 + 2.1x - 16.96$.

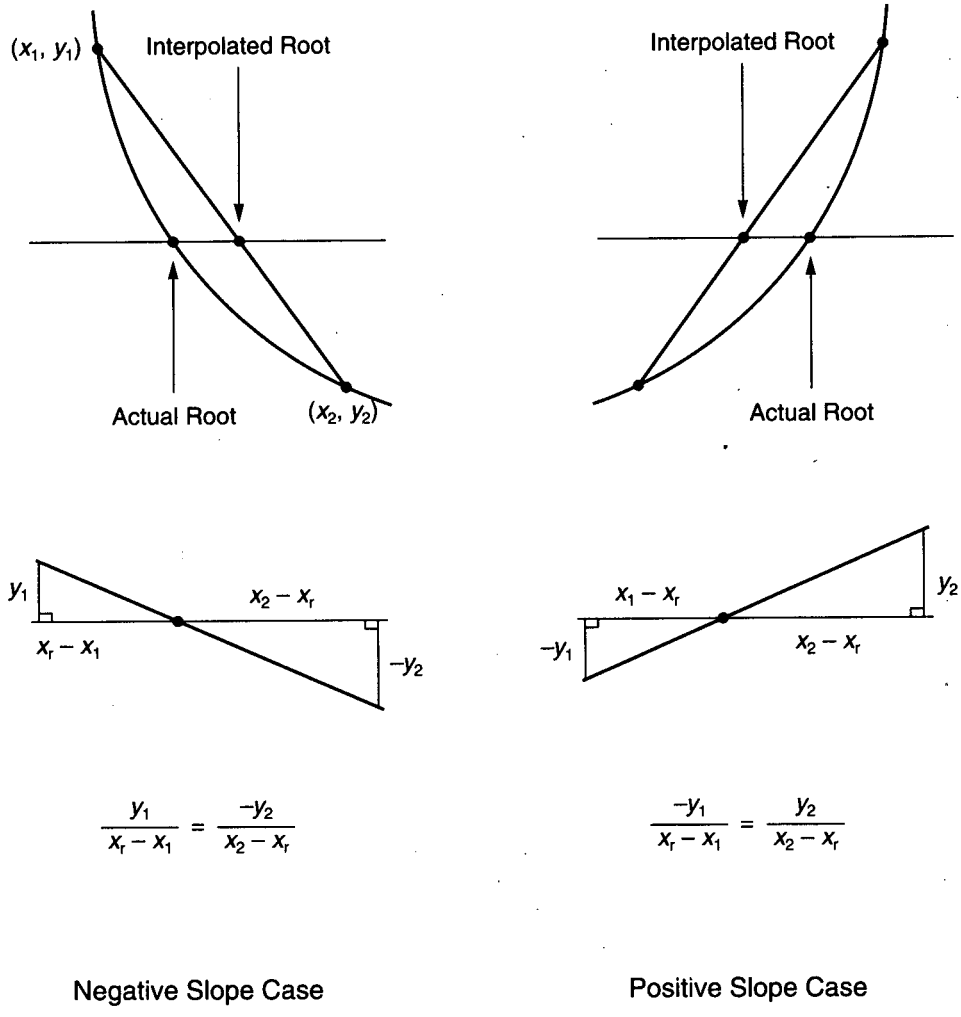


FIGURE 9–3 Approximating a Root by Interpolation

Following are four sample runs of Program 9-7 using different step sizes:

```
Enter the step size: 1
A root is at x = -5.276404
A root is at x = 3.182418

Enter the step size: 0.5
A root is at x = -5.292857
A root is at x = 3.193023

Enter the step size: 0.1
A root is at x = -5.300000
A root is at x = 3.200000

Enter the step size: 0.01
A root is at x = -5.300000
A root is at x = 3.200000
```

Program 9-7

```
#include <stdio.h>
#include <math.h>
main( )
{
  float f(float);      /* function prototype */
  float incr, x, x1, x2, y1, y2, xroot, yratio;

  printf("Enter the step size: ");
  scanf("%f", &incr);
  for ( x = -10; x <= 10 - incr; x += incr)
  {
    x1 = x;
    x2 = x + incr;
    y1 = f(x1);
    y2 = f(x2);
    if (y1 * y2 < 0.0)
    {
       yratio = y1 / (y1 - y2);
       xroot = x1 + yratio * (x2 - x1);
       printf("A root is at x = %f\n", xroot);
    }
  }
}
/* the function is evaluated below */
float f(float x)
{
return( pow(x,2.0) + 2.1 * x - 16.96);
}
```

The three important points to observe from Program 9-7 are:

1. The calculation of y values for the curve is relegated to a function. The expression computed and returned by this function can easily be changed for different curves.
2. Although two values of x (`x1` and `x2`) and two values of y (`y1` and `y2`) need to be calculated inside the loop, care is taken not to change the loop's controlling value, x, internal to the `for` loop. The maximum index value is set to `10-incr` to ensure that `x2` does not exceed 10.
3. Since the two values `y1` and `y2` must be of different sign for a root to exist, simply multiplying them and examining the sign of the result provides a convenient test. If both values are positive or if both are negative, the product is a positive number and the statements within the `if` statement are not executed.

We see from the results that by testing for a sign change and interpolating to find the root, the accuracy is quite good and the danger of missing a root has

been diminished. Even incrementing x by 1 each time results in roots that are less than 1 percent in error from their true value. For step sizes of 0.1 and lower, the roots rounded to 6 fractional digits are identical to the expected values of −5.3 and 3.2. Clearly, however, a root will be missed if the range of x values used in the `for` loop does not include the root.

The Bisection (Binary Search) Method

All fixed increment root-finding algorithms require the computer to iterate through a fixed number of x values, calculate corresponding y values, and qualify the results in some manner. When the step size is small this can be extremely time consuming, especially when the general vicinity of the roots is not known. In such cases it may be necessary to search over a wide range of x values to avoid missing a root. For reasons of computational efficiency, various techniques have been developed to permit the search to proceed much more rapidly than simply incrementing x by a fixed step size. One such method is the bisection method.

The rationale for the bisection method can be better understood by first considering a game called High-Low. In High-Low your opponent is asked to select an integer between 1 and 99, and you are required to find the number with as few guesses as possible. The game is often played against a calculator or computer, in which case you must guess an integer that is randomly selected by the machine. Upon each guess, you are told whether your choice is too high or too low. When the correct number is chosen, the game is over and the number of guesses is tallied.

If we approach this game as a fixed-step iteration problem, we would begin by selecting 1 as our first guess. If that is incorrect we will be told that our guess is too low. We would then begin incrementing our guess by 1 each time, proceeding to guess the numbers 2, 3, 4, . . . etc., until we located the right number. Clearly the number of tries is going to equal the number selected by our opponent. If a low number was selected we are in luck, but in the worst case it could take 99 guesses.

There is, however, a better approach to the game. Rather than select 1 as our first guess let us select a number at the middle of the range, namely 50. Now if we are told that the guess is "too low" we have instantly eliminated all of the integers below 50, as well as 50 itself. Conversely a response of "too high" removes from further consideration the numbers in the range 50 to 99. In either case the range has been cut approximately in half. Our next guess repeats this strategy: If the new range is 1 to 49, we choose 25, and if the new range is 51 to 99, we choose 75. In either case we again eliminate half of the remaining numbers (assuming that we haven't been so fortunate to hit the number already). You can see that this process will locate the correct number quite rapidly. An illustrative example is shown below:

Number selected by opponent: 59

Range	Guess	Message
1–99	50	Too low
51–99	75	Too high
51–74	63	Too high
51–62	57	Too low
58–62	60	Too high
58–59	58	Too low
59	59	Correct

Notice that the correct number was identified in 7 guesses, without any "lucky" guesses; that is, the number was not identified until it was the only possible one left. This is always the case: The maximum number of guesses required by halving each range is seven, regardless of the number originally selected (recall that the fixed-step iteration method might require 99 guesses).

Let us now apply the underlying strategy of High-Low to root finding. To do this consider a curve that is known to have a single root, denoted as x_r, between the values $x = a$ and $x = b$, as illustrated in Figure 9–4a. Although the illustrated curve has a positive slope, the technique works equally well with a curve having a negative slope. Regardless of the curve's slope the value of the curve at $x = a$ differs in sign from the value at $x = b$.

In the bisection method we initially calculate y values corresponding to the left and right bounds on the x-axis, namely $x = a$ and $x = b$. Then we select the midpoint of these two x values and replace one of the x values by this midpoint so that the root lies in the remaining range. For this procedure the midpoint of the values $x = a$ and $x = b$ is calculated as:

$$\text{mid} = (a+b) / 2.0$$

If the calculated midpoint lies to the left of the root, as is the case in Figure 9–4a, the left half of the curve segment (between $x = a$ and $x =$ mid) can be discarded; otherwise, the right half between $x =$ mid and $x = b$ can be removed from further consideration. In either case the root is within the reduced interval. Figure 9–4b shows the removal of the left half of the curve segment, with the value of a replaced by the previously calculated value of mid. The second calculation of mid, using this new value of a, results in a value larger than the root. In this case the right half of the remaining segment is discarded and the value of b is replaced by the newly calculated mid. Figure 9–4c shows the original segment reduced to 1/4 of its size. A third calculation of mid is seen to be smaller than the root, so that the next step would be to replace a by mid.

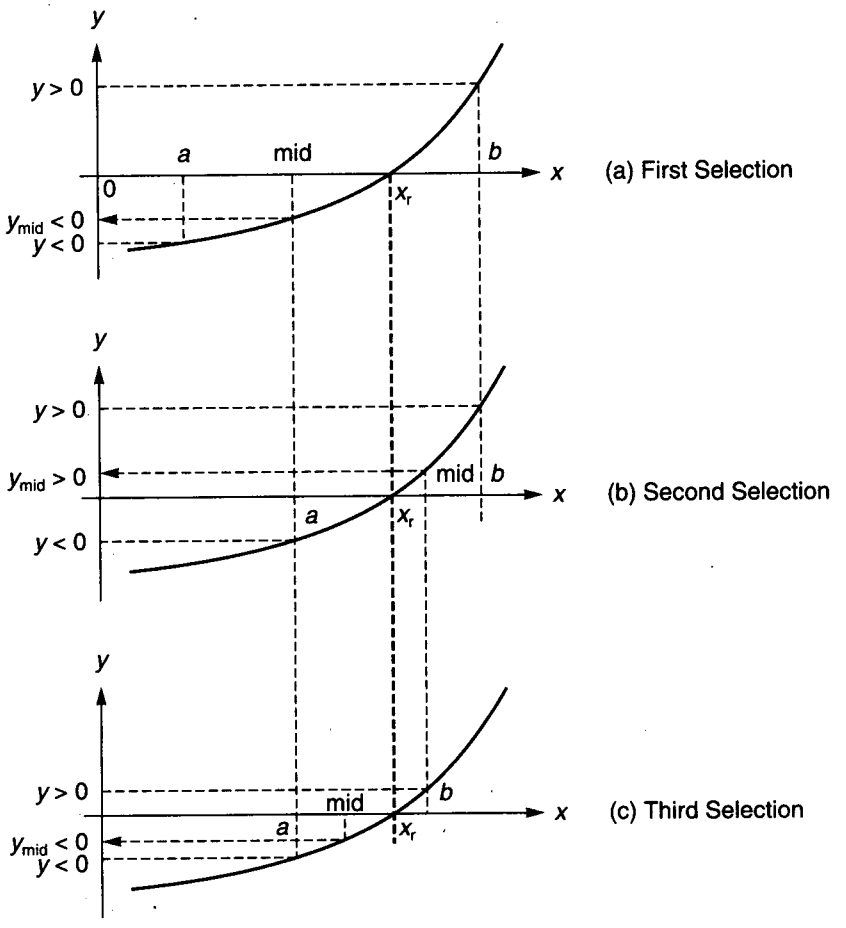
We see that each step of this procedure reduces the interval between $x = a$ and $x = b$ by a factor of two, or in other words bisects the interval. Now we can see the analogy of this procedure with the High-Low game. The initial values a and b correspond to the initial range of integers in the High-Low game, which was 1 to 99 in our illustration. The calculation of mid corresponds to the "guessing" of a number at the middle of the range. If mid is larger than the root the "guess" was too high and the upper half of the range is eliminated; alternatively, a low "guess" of mid, which is less than the root, removes the lower half of the interval from further consideration.

Since the High-Low game involves only integers an exact answer will always be obtained after a relatively small number of guesses. Because the roots of equations are generally real numbers subject to roundoff error, an exact answer, regardless of the root-finding method, generally is not possible. We must, therefore, agree beforehand on an allowable error in the computation of the root. We can also provide further control on the computation time by limiting the number of iterations as well. If we specify too small an error and too few iterations we run the risk of not finding the root. But if that happens we can then adjust one (or both) of these criteria and rerun the program. With this as background, the pseudocode for the bisection algorithm is as follows:

enter an allowable error for the root
and the maximum number of iterations

```
initialize left and right bounds of x
repeat until acceptable root found or maximum iterations is reached
   calculate midpoint of current interval
   if function value at left bound and at midpoint differ
      in sign then
         set new right bound to midpoint value
   else
      set new left bound to midpoint value
   endif
   if within the allowable error
      display root value and number of iterations
      set found flag
      stop
   endif
end repeat loop
if found flag not set
   display message indicating no root found
endif
```

FIGURE 9–4 Illustration of the Bisection Technique



Program 9-8 implements this pseudocode for the curve $y = x^2 + 2.1x - 16.96$.

Program 9-8

```
#include <stdio.h>
#include <math.h>
main( )
{
  int i, n, found = 0;
  float a, b, x, xerror, mid, xroot;
  float f(float);   /* function prototype */

  printf("Enter the maximum root error: ");
  scanf("%f", &xerror);
  printf("Enter the maximum number of iterations: ");
  scanf("%d", &n);
  printf("Enter the left and right bounds of x (a b): ");
  scanf("%f %f", &a, &b);
  for ( i = 1; i <= n; ++i)
  {
    mid = (a + b) / 2.0;
    if ( f(a) * f(mid) < 0.0)
      b = mid;
    else
      a = mid;
    if ( b - a <= xerror)
    {
      xroot = (a + b) / 2.0;
      printf("A root is at x = %f\n", xroot);
      printf("This root was found after %d iterations.\n", i);
      found = 1;
      break;
    }
  }
  if (!found) printf("No root was found after %d iterations\n", n);
}
/* the function is evaluated here */
float f(float x)
{
  return( pow(x,2.0) + 2.1 * x - 16.96 );
}
```

Notice that Program 9-8 uses a `for` loop that contains an `if` statement to terminate the program if a root is found. The actual termination is caused by the `break` statement, which is executed after the root value and number of iterations is printed. Also notice that the value of the root is computed by averaging the

last values of a and b, which yields a more accurate approximation to the root than either a or b alone.

A disadvantage of the bisection procedure is that two roots within an interval can cause problems because no sign change in y values will occur (can you see why?). The solution to this problem is to select an interval that encloses only one root, and then rerun the program for the additional roots. The sample runs shown below for Program 9-8 illustrate how both roots of the curve $y = x^2 + 2.1x - 16.96$ are found by modifying the range of the search interval.

```
Enter the maximum root error: 0.0001
Enter the maximum number of iterations: 30
Enter the left and right bounds of x (a b): -100 100
A root is at x = -5.299997
This root was found after 21 iterations.

Enter the maximum root error: 0.0001
Enter the maximum number of iterations: 30
Enter the left and right bounds of x (a b): 0 100
A root is at x = 3.200006
This root was found after 20 iterations.
```

In the first run the program correctly locates the root at $x = 5.3$, but misses the positive root. The next run locates the positive root at $x = 3.2$ because the search interval includes this root while excluding the negative root.

Secant Method

The secant method is a variation on the bisection technique just described. Rather than selecting the midpoint of an interval at each step to estimate the root value, the intersection of a secant line with the x-axis is used as the next estimate. A *secant line* is a straight line that connects two points on the curve, as illustrated in Figure 9–5.

Figure 9–5 also shows the process of estimating the root location through the use of successive secant lines. The initial interval of the curve is defined by the values x_1 and x_2; notice that these points correspond to points a and b respectively for the bisection method. In Figure 9–5a the first secant line drawn crosses the x-axis at x_3; for the curve shown it is to the left of the actual root but only because of the bending of the curve.

A second secant line shown in Figure 9–5b connects the points of the curve corresponding to x_2 and x_3; the intersection of this secant line with the x-axis is then x_4. Again the last two values of x, namely x_3 and x_4, define the points on the curve corresponding to the next secant line. As shown in Figure 9–5c the third secant line intersects the x-axis at x_5. The process would then continue, using next x_4 and x_5 to define the next curve segment.

Notice that the point x_5 in this illustration falls outside of the interval between x_3 and x_4. This is one difference between this method and the bisection method where each point chosen is halfway between the two previous points. The fact that a point may fall outside the interval defined by the previous two points does not imply that the secant method is diverging. In fact for most cases it converges more rapidly than does the bisection method, which means that the intersections of the secant lines with the x-axis are more rapidly approaching the actual root

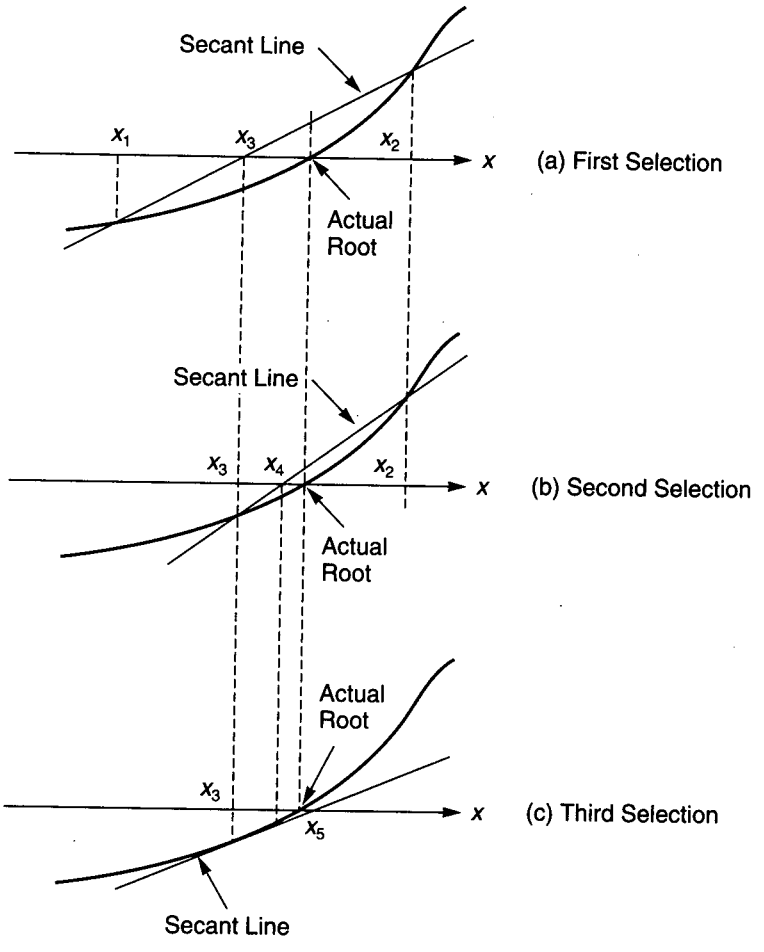


FIGURE 9–5 Illustration of Root Finding using the Secant Method

location. On the other hand, as with the bisection method there are potential problems with this method when multiple roots are involved.

A program to implement the secant method is no more complex than Program 9-8, which implemented the bisection method. The secant method requires repeated calculation of the intersection of the line with the x-axis in place of the simpler calculation of the midpoint when bisection is used. However, the bisection method includes for each step a test of whether the function value at the midpoint has the same sign as the value at one of the endpoints; the `if-else` statement within the `for` loop of Program 9-9 implements this test. The test is necessary to determine whether the left or right end of the previous interval is used as a bound for the next interval. Since the secant method always uses the two most recent values of `x` to define the next interval, regardless of the value of the function at any point, the test is unnecessary.

We can terminate the secant method's root search when the function's value for the most recent value of `x` is arbitrarily close to zero. As was done for the bisection method we can terminate the program before finding a root when a desired maximum number of iterations have been completed.

The pseudocode for the secant method is the following:

enter acceptable error for root and maximum iterations
initialize left and right bounds of x
repeat until acceptable root found or maximum iterations is reached
  calculate location of secant line intersection with
    x-axis for current bounds
  use secant intersection value to update bound locations
display root value and number of iterations, or message
indicating no root found

Recall the calculation of the secant line's intersection with the x-axis shown in Figure 9–3 and implemented in Program 9-7. Notice that the result of the single calculation of x_r from x_1 and x_2 in Figure 9–3 can be applied to the repeated calculation of x_3, x_4, x_5, . . . illustrated in Figure 9–5. We are now finding x_{i+2} from x_i and x_{i+1}, starting with $i = 1$, where the first two values x_1 and x_2 are the initial bounds to be entered into our program.

The relationship

$$x_r = x_1 + \frac{y_1}{y_1 - y_2}(x_2 - x_1)$$

previously shown now becomes

$$x_{i+2} = x_i + \frac{y_i}{y_i - y_{i+1}}(x_{i+1} - x_i) \qquad (i = 1, 2, 3, \ldots)$$

where x_{i+2} is the location of the intersection of the current secant line with the x-axis, x_{i+1} and x_i are the previously calculated secant line intersections, and y_{i+1} and y_i are the function values corresponding to x_{i+1} and x_i respectively.

Program 9-9 implements the secant method for the same parabolic function used previously, namely:

$$x^2 + 2.1x - 16.96$$

It is convenient to use single-dimension arrays in this program to repeatedly calculate the values of `x` and `y`. Arrays `x` and `y` have 102 memory locations assigned to them as a result of their declaration statement. This allows us to perform as many as 100 iterations, noting that the first two array elements are reserved for the values entered into `x[0]` and `x[1]` and calculated for `y[0]` and `y[1]`. Notice that the variable name `x` in the function `f( )` does not conflict with the array named `x` in the `main( )` function.

Another feature to notice is the `printf( )` function call inside the `for` loop. Its purpose is simply to display the result of each calculation of the secant line intersection with the x-axis. This is a useful aid in debugging such a program, and offers some insight into the degree of convergence of this method; it can, of course, be eliminated from the program.

Program 9-9

```
#include <stdio.h>
#include <math.h>
main( )
{
  int i, n;
  float yerror, xroot, yratio;
  float x[102], y[102];
  float f(float);   /* function prototype */

  printf("Enter the maximum function error: ");
  scanf("%f", &yerror);
  printf("Enter the maximum number of iterations ( <=100 ): ");
  scanf("%d",&n);
  printf("Enter left and right bounds of x (x[0], x[1]): ");
  scanf("%f %f", &x[0], &x[1]);
  for ( i = 0; i <= n-1; ++i)
  {
    y[i] = f(x[i]);
    y[i+1] = f(x[i+1]);
    yratio = y[i] / (y[i] - y[i+1]);
    x[i+2] = x[i] + yratio * (x[i+1] - x[i]);
    printf("\nx[%d] = %12.6f", i+2, x[i+2]);
    y[i+2] = f(x[i+2]);
    if ( fabs(y[i+2]) <= yerror)
    {
      xroot = x[i+2];
      printf("\nA root is at x = %f", xroot);
      printf("\nRoot found after %d iterations", i+1);
      break;
    }
  }
}
/* the function is evaluated here */
float f(float x)
{
  return( pow(x,2.0) + 2.1 * x - 16.96 );
}
```

Several sample runs of Program 9-9 are shown below along with comments on the results. For each run the acceptable function error is entered as 0.0001 and a 30-iteration limit is specified.

```
Enter the maximum function error: 0.0001
Enter the maximum number of iterations ( <=100 ): 30
Enter left and right bounds of x (x[1], x[2]): -100 100
```

```
x[2] = -4753.828613
x[3] =   102.191238
x[4] =   104.479561
x[5] =    51.223000
x[6] =    34.021729
x[7] =    20.146093
x[8] =    12.482533
x[9] =     7.729481
x[10] =    5.084414
x[11] =    3.772313
x[12] =    3.298430
x[13] =    3.206143
x[14] =    3.200070
x[15] =    3.200000
A root is at x = 3.200000
Root found after 14 iterations
```

When the same values were entered for the bisection program the smaller root $x = -5.3$ was found after 21 iterations. Here the larger rather than the smaller root was found but with fewer iterations. Notice that the large range specified results in the first calculation of `x[3]` being far removed from the root. This is not surprising in view of the fact that the parabolic function evaluates to about 10,000 for $x = -100$ and for $x = 100$.

```
Enter the maximum function error: 0.0001
Enter the maximum number of iterations ( <=100 ): 30
Enter left and right bounds of x (x[1], x[2]): 100 -100
x[2] = -4753.828613
x[3] =   -97.985657
x[4] =   -96.051971
x[5] =   -49.123577
x[6] =   -33.097034
x[7] =   -20.504147
x[8] =   -13.506224
x[9] =    -9.209971
x[10] =   -6.856354
x[11] =   -5.735712
x[12] =   -5.364632
x[13] =   -5.303129
x[14] =   -5.300024
x[15] =   -5.300000
A root is at x = -5.300000
Root found after 14 iterations
```

For this run we "fool" the program by reversing the bound numbers entered; despite the prompt message the right bound is entered before the left bound and the program has no provision for rejecting the entry. Notice that the first calculation is the same as for the previous run; a secant line is drawn between the same two points regardless of which one we choose as `x[1]`. The next calculation, that of `x[4]`, and subsequent ones differ from those from the first run (can you see

why?). The result is that the negative root rather than the positive root has been found in the same number of iterations.

```
Enter the maximum function error: 0.0001
Enter the maximum number of iterations ( <=100 ): 30
Enter left and right bounds of x (x[1], x[2]): -100 0
x[2] =    -0.173241
x[3] =     8.802341
x[4] =     1.438618
x[5] =     2.400397
x[6] =     3.437145
x[7] =     3.176111
x[8] =     3.199350
x[9] =     3.200002
A root is at x = 3.200002
Root found after 8 iterations
```

This result illustrates a problem with the secant method when more than one root exists. Only the negative root is enclosed within the initial bound (−100 to 0), yet the positive root is the one found. The problem, as we saw for the curve shown in Figure 9–5, is that the last two secant line intersections with the x-axis don't always stay between the root we are searching for. As a result the process can converge to another root, as is the case for this run.

```
Enter the maximum function error: 0.0001
Enter the maximum number of iterations ( <=100 ): 30
Enter left and right bounds of x (x[1], x[2]): -10 0
x[2] =    -2.146835
x[3] =  -362.126465
x[4] =    -2.193386
x[5] =    -2.239640
x[6] =    -9.375125
x[7] =    -3.989264
x[8] =    -4.825814
x[9] =    -5.392558
x[10] =    -5.294594
x[11] =    -5.299942
x[12] =    -5.300000
A root is at x = -5.300000
Root found after 11 iterations
```

For this run we chose the left interval as −10 rather than −100 and the negative root was found. However, the convergence was not as rapid as we might have expected since an apparently small slope for the second secant line constructed resulted in a large negative value for `x[4]`. This again illustrates that this method is rather sensitive to the nature of the curve.

```
Enter the maximum function error: 0.0001
Enter the maximum number of iterations ( <=100 ): 30
Enter left and right bounds of x (x[1], x[2]): -6 -4
```

```
x[2] =     -5.184810
x[3] =     -5.321136
x[4] =     -5.299710
x[5] =     -5.299999
A root is at x = -5.299999
Root found after 4 iterations
```

For this run it is assumed that we had some knowledge that the negative root was in the vicinity of $x = -5$. By choosing the initial interval very narrow in comparison to the previous runs we converged to the enclosed root quite rapidly.

From the results of these runs we can conclude that the secant method does indeed converge more rapidly than the bisection method in most cases, and certainly far more rapidly than the fixed-count iteration method presented at the beginning of this section. However, where more than one root exists it is not always apparent as to how to locate all the roots with a very small number of runs. You may want to try other functions with this method in order to gain more insight into the problem.

Exercises 9.2

1a. Modify Program 9-4 to calculate and display a table of integer values of the function

$$y = x^4 + 4x^3 - 7x^2 - 22x + 24$$

in the interval from $x = -10$ to $x = 10$. Compile and run the program on a computer. Can you determine the roots of this function from your display?

b. Modify Program 9-5 to calculate and display the roots of this function. Compile and run the program.

2. Modify Program 9-6 using the function from Exercise 1. Run the program four times using a step size of 0.01 for each run and allowable errors of 0.1, 0.01, 0.003, and 0.001 respectively.

3. Modify Program 9-7 using the function from Exercise 1. Run the program five times using step sizes of 1, 0.5, 0.499, 0.1, and 0.01 respectively. Explain your results.

4. Consider the function $y = \cos(x)$ in the interval from $x = 0$ to $x = 20$.

a. Sketch the curve for the function by hand and determine from it the number of roots in this interval and their location.

b. Modify Program 9-6 using this function and run the program. By trial and error set a step size and allowable error that will display each root in the interval only once.

5. Modify Program 9-7 using the function from Exercise 4. Run the program four times using step sizes of 1, 0.5, 0.1, and 0.01 respectively. Explain your results.

6. Modify Program 9-8 using the function from Exercise 1. Run the program several times, entering 0.001 for the allowable error and 30 for the maximum number of iterations. For the various runs select left and right bounds to enable you to find all the roots in the interval from $x = -10$ to $x = 10$.

7. Repeat Exercise 6 using the function $y = \cos(x)$ in the interval from $x = 0$ to $x = 20$.

8. Modify Program 9-9 using the function from Exercise 1. Run the program several times, entering 0.001 for the maximum function error and 30 for the maximum number of iterations. For the various runs select left and right bounds to enable you to find all the

roots in the interval from $x = -10$ to $x = 10$. Compare the number of iterations required for the secant method and the bisection method. Did the presence of multiple roots cause any problems in using the secant method?

9. Repeat Exercise 8 using the function $y = \cos(x)$ in the interval from $x = 0$ to $x = 20$.

10. The equation

$$2e^{-x} - 1 = 0$$

has a root between $x = 0$ and $x = 1$.

a. Modify and run Program 9-8 to locate the root using the bisection method, using the interval from 0 to 1 and an allowable error of 0.001. Have your program display the number of iterations made. Then modify Program 9-9 to locate the root using the secant method, using the same allowable error. Compare the number of iterations required by both programs.

b. Repeat Exercise 8a using an interval from 0 to 5. Determine the number of iterations required to locate the root using both the bisection and secant methods.

c. Repeat Exercise 8a using an interval from 0 to 10. Determine the number of iterations required to locate the root using both the bisection and secant methods. (Note that convergence may be a problem with the secant method.)

9.3 Numerical Integration

Integration is a calculus technique that can be used to find the area under a portion of a curve. In engineering and statistical problems the calculated areas frequently correspond to physical quantities. For example, the area under a normal bell-shaped curve in statistical applications is used in calculating probabilities and the area under a band of frequencies in engineering applications is used to calculate power consumption. As in the root-finding methods discussed in the previous section, there are approximation methods that can be easily programmed on a computer for determining these areas when exact solutions do not exist.

Rectangular Approximations

To illustrate the rectangular approximation method, consider Figure 9–6. As illustrated in this figure, an arbitrary function $y = f(x)$ is shown within an interval bounded by $x = a$ and $x = b$. The area under the curve refers to the area between the curve and the horizontal x-axis, bounded by the vertical lines drawn at a and b. An approximation to the true area under the curve can be obtained by dividing the area into n rectangles of equal widths, denoted as w in the figure, and adding the area of each rectangle. The integer n is arbitrary, but larger values chosen for n generally result in more accurate approximations. The width of each subinterval is the total interval width divided by the number of subintervals. Therefore, we have

$$w = \frac{b - a}{n}$$

The start of each subinterval is designated by $x_1, x_2, x_3, \ldots x_n$. Using a variable index, i, the start of subinterval i is at

$$x_i \qquad \text{where } i = 1, 2, 3, \ldots, n$$

The start of the first subinterval, x_1, corresponds to a and the end of the nth subinterval, which we can call x_{n+1}, is equal to b. Also, since the subintervals are of equal width,

$$x_{i+1} = x_i + w \qquad \text{for } i = 1, 2, 3, \ldots, n$$

The total area under the curve between a and b is clearly the sum of the areas under the n subintervals. As the name implies this first technique simply approximates the area under each subinterval by that of a rectangle. As seen in the enlarged drawing of the ith subinterval in Figure 9–6, the value of y at the start of the interval determines the height of the rectangle. Since the area of a rectangle is simply width multiplied by height, and the width is w, the area of the ith subinterval can be expressed as:

$$A_i = w * f(x_i)$$

This corresponds to the shaded area shown in Figure 9–6. The total area is then

$$\begin{aligned} \text{Area} &= A_1 + A_2 + A_3 + \ldots + A_n \\ &= \sum_{i=1}^{n} A_i \\ &= w \sum_{i=1}^{n} f(x_i) \end{aligned}$$

$$\text{where } w = \frac{b-a}{n}, \quad x_1 = a, \quad \text{and } x_{i+1} = x_i + w$$

for all i from 1 to n. This result approximates the area under the curve between limits $x = a$ and $x = b$.

This technique can be programmed with the aid of a looping procedure that calculates each subinterval area and then updates the value of x. The pseudocode for this procedure is given by:

```
enter left and right bounds (a and b) and the number of subintervals (n)
calculate width (w)
initialize x
set total area to zero
repeat for all subintervals
   calculate rectangle area and add to total area
   update value of x
end repeat loop
display total area
```

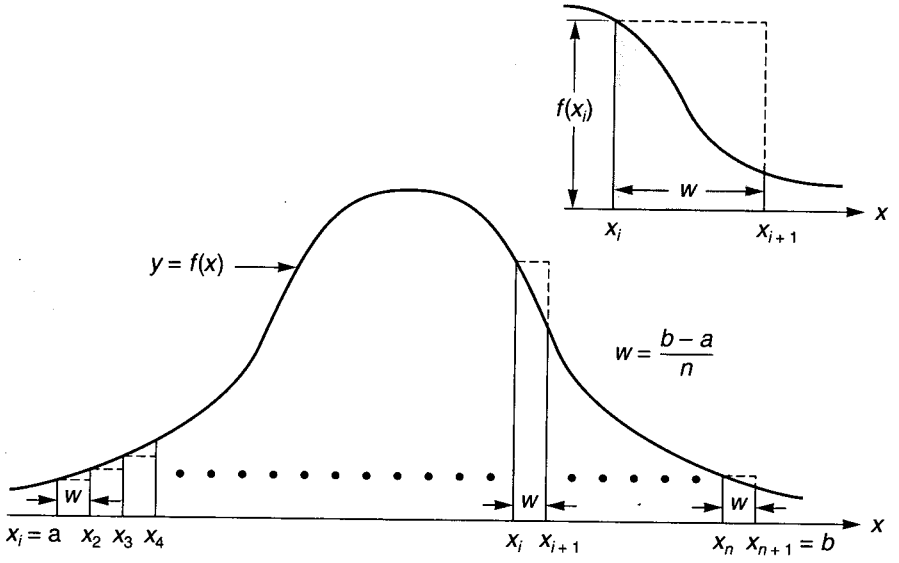


FIGURE 9–6 Rectangular Approximation to Area under a Curve

To illustrate this approximation method we will choose a curve for which the exact area is known, so that we may compare the calculated approximation to the actual area. Figure 9–7 shows a circle centered at the origin of the *x-y* coordinate system. Consider the area under the portion of the circle in the upper right quadrant, shown by the shaded region in Figure 9–7. From symmetry the shaded area is one-fourth of the total area of the circle, which we know to be π multiplied by the radius squared (where π = 3.141593, accurate to six decimal places). Therefore, the exact area of the shaded area is:

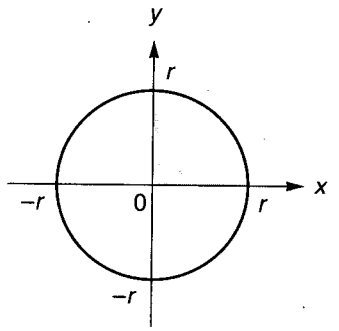
$$\text{Area} = \pi * r^2 / 4$$

For a circle of radius equal to 2 the area is then π, or 3.141593.

The equation of a circle centered at the origin is given by:

$$x^2 + y^2 = r^2$$

FIGURE 9–7 Area under a Circle



Equation of the Circle

$x^2 + y^2 = r^2$

For a circle with a radius of 2, this becomes

$$x^2 + y^2 = 4$$

Solving for y in terms of x, we obtain

$$y = \sqrt{4 - x^2}$$

To limit ourselves to the upper right quadrant requires that x be bounded by 0 and 2 and the positive (rather than negative) square root taken for y. The requirement on the sign of the root is no problem because C's square root function `sqrt( )` always returns the positive root.

Program 9-10 calculates and displays the area of this circle using rectangular approximation.

Program 9-10

```
#include <stdio.h>
#include <math.h>
main( )
{
  int i, n;
  float x, a, b, width, area;
  float f(float);   /* function prototype */

  printf("Enter the left and right bounds of x (a b): ");
  scanf("%f %f", &a, &b);
  printf("Enter the number of subintervals (n): ");
  scanf("%d", &n);
  width = (b - a) / n;
  x = a;
  area = 0;
  for ( i = 1; i <= n; ++i)
  {
    area = area + width * f(x);
    x = x + width;
  }
  printf("The approximate total area using\n ");
  printf(" the rectangular method is: %f", area);
}
/* the function is evaluated below */
float f(float x)
{
  return( sqrt(4.0 - pow(x,2.0)));
}
```

Notice that although we used subscripted x values in the discussion of the method, the difference between x_{i+1} and x_i is a constant for all i. It is therefore unnecessary to use arrays in the program to store many values of x at once. After each value is used in calculating the area of the subinterval it is updated by the statement `x = x + width`. This calculation in effect replaces each x_i by the next value, x_{i+1}. Notice also that the index `i` in the program simply functions as a counter, allowing the areas of the `n` subintervals to be summed. The equation of the circle is specified by the function `f( )`.

The sample runs for Program 9-10 are shown below, varying only the number of subintervals, `n`.

```
Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 2
The approximate total area using
 the rectangular method is: 3.732051

Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 10
The approximate total area using
 the rectangular method is: 3.304518

Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 50
The approximate total area using
 the rectangular method is: 3.178269

Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 1000
The approximate total area using
 the rectangular method is: 3.143580
```

As expected the accuracy improves as `n` gets larger. For example, if only two subintervals are used the approximated area is about 20 percent greater than the exact area (π = 3.141593, accurate to six decimal places); however, if 50 subintervals are used the error is about 1 percent. Obviously the improved accuracy is gained at the expense of a longer running time for the program.

Modified Rectangular Approximations

A modified form of the rectangular method, called the *midpoint method*, is illustrated in Figure 9–8. The only difference from the standard rectangular method is that the value of y at the midpoint of the subinterval, rather than at the start of the subinterval, is used as the height of the rectangle.

From Figure 9–8 we see that at the midpoint of the ith rectangle m_i is calculated as:

$$m_i = \frac{x_i + x_{i+1}}{2} = \frac{x_i + (x_i + w)}{2} = x_i + \frac{w}{2}$$

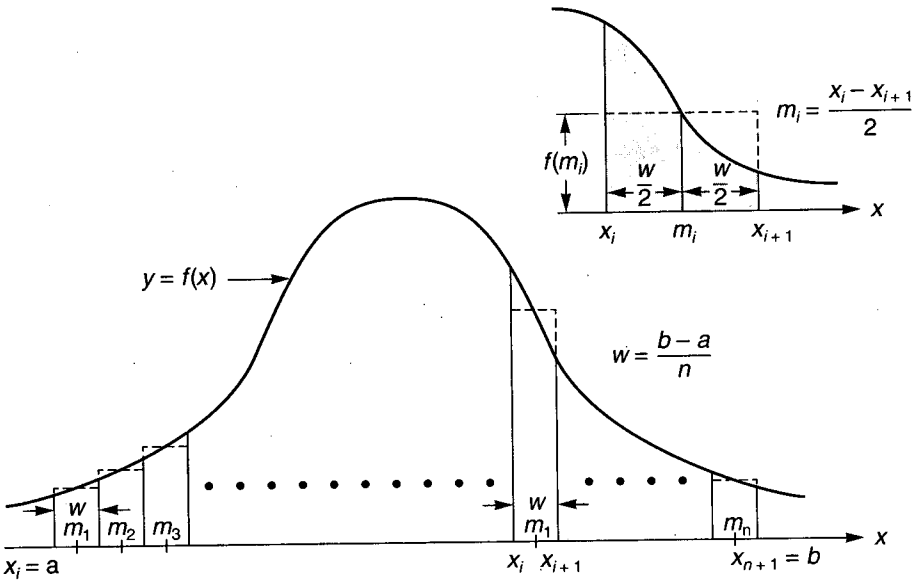


FIGURE 9–8 Modified Approximation to Area under a Curve

where w is the width of each rectangle. The height of the ith rectangle is the y value for $x = m_i$, which is:

$$f(m_i) = f\left(x_i + \frac{w}{2}\right)$$

and the area of the ith rectangle is then

$$A_i = w * f\left(x_i + \frac{w}{2}\right)$$

Finally, the total area under the curve is approximated by

$$\text{Area} = w\sum_{i=1}^{n} f\left(x_i + \frac{w}{2}\right)$$

where the width, w, of each rectangle is calculated as $(b - a) / n$.

From a programming perspective, this approximation formula for the total area can be programmed using a `for` loop that calculates each rectangular area, adds the calculated area to a sum, and then updates the value of x. The pseudocode for this procedure is given by

enter the left and right bounds (a and b) and
the number of subintervals (n).
calculate width (w)
initialize x
set total area to zero
repeat for all subintervals

calculate the x midpoint and its corresponding y value
calculate the rectangle area and add to the total area
update the value of x
end repeat loop
display total area

Program 9-11 calculates and displays the area of this circle using the midpoint approximation method.

Program 9-11

```
#include <stdio.h>
#include <math.h>
main( )
{
  int i, n;
  float x, a, b, width, area;
  float f(float);   /* function prototype */

  printf("Enter the left and right bounds of x (a b): ");
  scanf("%f %f", &a, &b);
  printf("Enter the number of subintervals (n): ");
  scanf("%d", &n);
  width = (b - a) / n;
  x = a;
  area = 0;
  for ( i = 1; i <= n; ++i)
  {
    area = area + width * f(x + width/2.0);
    x = x + width;
  }
  printf("The approximate total area using\n ");
  printf(" the midpoint method is: %f", area);
}
/* the function is evaluated below */
float f(float x)
{
  return( sqrt(4.0 - pow(x,2.0)));
}
```

Notice that after each midpoint value is used to calculate a corresponding `y` value and the area of each rectangle is computed, this area is immediately added into the total area. Notice also that the variable `i` in the program only acts as a counter, allowing the areas of the `n` rectangles to be summed. The equation of the circle is specified by the function `f( )`.

Four sample runs using Program 9-11 follow, showing the areas calculated when 2, 10, 50, and 1000 rectangles are used.

```
Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 2
The approximate total area using
 the midpoint method is: 3.259367

Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 10
The approximate total area using
 the midpoint method is: 3.152411

Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 50
The approximate total area using
 the midpoint method is: 3.142566

Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 1000
The approximate total area using
 the midpoint method is: 3.141631
```

As expected, the accuracy improves as the number of approximating rectangles, n, is increased. Using only two rectangles the approximation differs by less than 5 percent from the exact area (3.141593). This compares to a 20 percent difference using the standard rectangular method. For $n = 50$, the error is less than 0.01 percent of its true value, which compares to an error of 1 percent using the same number of subintervals with the standard rectangular method. Referring to Figures 9–6 and 9–8 again, it seems reasonable to expect that using the midpoint for the height of each rectangle method would give a more accurate result than using the endpoint, as in the standard rectangular method.[2] If the curve does not exhibit unusual bending within a subinterval, the use of the midpoint to define the height of the rectangle will give a better approximation to the subinterval area than if the left endpoint of the subinterval (or the right endpoint, for that matter) were used.

Trapezoidal Approximation

In Figure 9–9, which illustrates the trapezoidal approximation method, the interval is divided into subintervals in the same manner as for the rectangular methods. However, the curve is now approximated by connecting a straight line between the points on the curve corresponding to the start and end of the subinterval. As shown by the enlarged drawing of the ith subinterval, the approximate area (the shaded portion) is that of a trapezoid made up of a rectangle plus a tri-

[2] Mathematically, this depends on the convexity/concavity of the particular curve. That is, the error is affected mainly by the second derivative of the function.

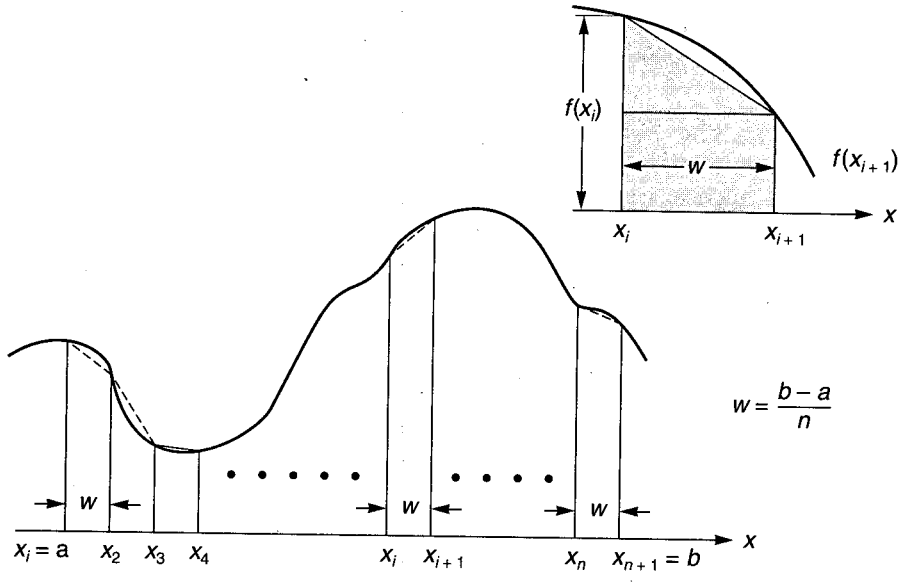


FIGURE 9–9 Trapezoidal Approximation to Area under a Curve

angle. Since the area of a triangle is 1/2 of the product of width and height, the area of the ith subinterval shown in Figure 9–9 is then

$$\begin{aligned} A_i &= \text{area of rectangle} + \text{area of triangle} \\ &= w\ f(x_{i+1}) + 0.5\ w\ [f(x_i) - f(x_{i+1})\ ] \\ &= 0.5\ w\ [f(x_i) + f(x_{i+1})\ ] \end{aligned}$$

The total area is:

$$\begin{aligned} \text{Area} &= A_1 + A_2 + A_3 + \ldots + A_n \\ &= 0.5\ w\ [f(x_1) + f(x_2)] \\ &\quad + 0.5\ w\ [f(x_2) + f(x_3)] \\ &\quad + 0.5\ w\ [f(x_3) + f(x_4)] \\ &\quad + \ldots\ldots \\ &\quad + 0.5\ w\ [f(x_n) + f(x_{n+1})] \\ &= 0.5\ w\ [f(x_1) + f(x_{n+1})] \\ &\quad + w\ [f(x_2) + f(x_3) + f(x_4) + \ldots + f(x_n)] \end{aligned}$$

Since x_1 and x_{n+1} are the interval endpoints a and b, respectively, we can write the total area as:

$$\text{Area} = 0.5w\left[f(a) + f(b)\right] + w\sum_{i=2}^{n} f(x_i)$$

$$\text{where} \quad w = \frac{b-a}{n} \quad \text{and} \quad x_{i+1} = x_i w$$

The computation of the total area can now be done in a manner similar to that done for the rectangular methods. After entering `a`, `b`, and `n` the area can be initialized to

```
0.5 w [ f(a) + f(b) ]
```

and the terms $w\ f(x_i)$ can be added within a loop. Notice that in contrast to the results for the rectangular methods the summation starts for `i = 2` rather than `i = 1`. The initial loop index value is therefore `2` for this case. Program 9-12 uses the trapezoidal approximation to compute the area of the quarter circle in Figure 9–7, with radius `r = 2`.

Program 9-12

```
#include <stdio.h>
#include <math.h>
main( )
{
  int i, n;
  float x, a, b, width, area;
  float f(float);   /* function prototype */

  printf("Enter the left and right bounds of x (a b): ");
  scanf("%f %f", &a, &b);
  printf("Enter the number of subintervals (n): ");
  scanf("%d", &n);
  width = (b - a) / n;
  x = a + width;
  area = 0.5 * width * ( f(a) + f(b) );
  for ( i = 2; i <= n; ++i)
  {
    area = area + width * f(x );
    x = x + width;
  }
  printf("The approximate total area using\n ");
  printf(" the trapezoidal method is: %f", area);
}
/* the function is evaluated below */
float f(float x)
{
  return( sqrt(4.0 - pow(x,2.0)));
}
```

As for the rectangular methods the four sample runs for the trapezoidal method shown below display the approximate area for 2, 10, 50, and 1000 subintervals respectively.

```
Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 2
The approximate total area using
 the trapezoidal method is: 2.732051
```

```
Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 10
The approximate total area using
 the trapezoidal method is: 3.104518

Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 50
The approximate total area using
 the trapezoidal method is: 3.138269

Enter the left and right bounds of x (a b): 0 2
Enter the number of subintervals (n): 1000
The approximate total area using
 the trapezoidal method is: 3.141580
```

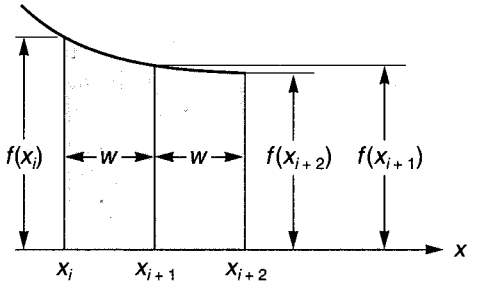
The results for this example show that the trapezoidal method is considerably more accurate than the standard rectangular method, but not quite as accurate as the modified rectangular method. Although we cannot generalize our conclusions for all curves, the use of the two endpoints of each subinterval in most cases should enable a better approximation of the area than the use of only one endpoint. This justifies the trapezoidal approach as compared to the standard rectangular method. The modified rectangular method, unlike the standard method, is actually using the two endpoint values of x in calculating the subinterval's midpoint. We would therefore expect comparable accuracy between the modified rectangular and trapezoidal methods. The results for the area of the quarter circle support that conclusion.

Simpson's Method

In addition to obtaining a more accurate estimate of the area under a curve by increasing the number of approximating rectangles, it is frequently possible to increase the accuracy of the approximation by using a better approximation for the height of each rectangle. Simpson's method effectively achieves this result by fitting a parabolic curve to the top of the rectangle rather than simply using a straight line. Figure 9–10 shows such a parabolic curve being fitted to the endpoints of two successive subintervals.

Notice that the two subintervals shown in Figure 9–10 share a common endpoint at x_{i+1}, providing three points to which the parabolic curve is being fit.

FIGURE 9–10 Parabolic Approximation to the Area under a Curve



Since two subintervals are being fitted at a time, the total number of subintervals, n, must be an even integer when Simpson's method is used.

The shaded area shown in Figure 9–10 is the approximation to the actual area under the curve between the two subintervals bounded by $x = x_i$ and $x = x_{i+2}$. The area under a parabolic curve cannot be determined simply as the height times the width, as for a rectangle. With the aid of calculus, however, it can be shown that the shaded area is equal to

$$A_i + A_{i+1} = \frac{w}{3} \cdot \left[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\right]$$

where A_i is the area of the interval starting at x_i and A_{i+1} the area of the adjacent next interval, starting at x_{i+1}. If the curve encompassed within two adjacent subintervals was flat (zero slope), then $f(x_i) = f(x_{i+1}) = f(x_{i+2})$, and the shaded area becomes two equal-sized rectangles, each of width w and height equal to $f(x_i)$. In this case the previous expression reduces to

$$\begin{aligned} A_i + A_{i+1} &= \frac{w}{3} \cdot \left[f(x_i) + 4f(x_i) + \right] \\ &= \frac{w}{3} \cdot \left[6f(x_i)\right] = 2wf(x_i) \end{aligned}$$

as expected.

The total area under the curve, between the limits $x = a$ and $x = b$, is again a summation of areas A_i for all i from 1 to n. However, since the expression above includes two adjacent subintervals the sum is taken for $i = 1, 3, 5$, and all other odd values of i up to and including $n-1$. That is,

$$\begin{aligned} \text{Area} &= \sum_{i=1}^{n} \left(A_i + a_{i+1}\right) \\ &\quad (\text{odd } i \text{ only}) \\ &= \frac{w}{3} \cdot \sum_{i=1}^{n} \left[f(x_i) + 4f(x_{i+1}) + f(x_{i=2})\right] \\ &\quad (\text{odd } i \text{ only}) \end{aligned}$$

Rather than expand this expression we will find it convenient to evaluate it in its present form using a `for` loop. As in Program 9-10, a routine to implement Simpson's method must input the values of a and b and the number of subintervals (n), and calculate the width. The calculation of the total area is then completed using the following statements:

```
x = a;
area = 0;
for ( i = 1; i < n; i += 2)
{
  temp = f(x) + 4.0 * f(x+width) +f(x + 2.0*width);
  area = area + width * temp / 3.0;
  x = x + 2*width;
}
```

Starting with the leftmost interval, x is initialized to a and area to 0. Within the loop the index advances by two each time and x is incremented by twice the width, to account for the handling of two subintervals at a time. Additionally, a temporary variable (temp) is introduced to avoid an overly lengthy statement line for the calculation of area.

Program 9-13 includes these statements in applying Simpson's method to approximate the area of the quarter circle with radius $r = 2$.

Program 9-13

```
#include <stdio.h>
#include <math.h>
main( )
{
  int i, n;
  float x, a, b, temp, width, area;
  float f(float);   /* function prototype */

  printf("Enter the left and right bounds of x (a b): ");
  scanf("%f %f", &a, &b);
  printf("Enter an even number of subintervals (n): ");
  scanf("%d", &n);
  width = (b - a) / n;
  x = a;
  area = 0;
  for ( i = 1; i < n; i += 2)
  {
    temp = f(x) + 4.0 * f(x+width) +f(x + 2.0*width);
    area = area + width * temp / 3.0;
    x = x + 2*width;
  }
  printf("The approximate total area using\n ");
  printf(" Simpson's method is: %f", area);
}
/* the function is evaluated below */
float f(float x)
{
  return( sqrt(4.0 - pow(x,2.0)));
}
```

The sample runs below show the results for 2, 10, 50, and 1000 subintervals:

```
Enter the left and right bounds of x (a b): 0 2
Enter an even number of subintervals (n): 2
The approximate total area using
 Simpson's method is: 2.976068
```

```
Enter the left and right bounds of x (a b): 0 2
Enter an even number of subintervals (n): 10
The approximate total area using
 Simpson's method is: 3.127008

Enter the left and right bounds of x (a b): 0 2
Enter an even number of subintervals (n): 50
The approximate total area using
 Simpson's method is: 3.140292

Enter the left and right bounds of x (a b): 0 2
Enter an even number of subintervals (n): 1000
The approximate total area using
 Simpson's method is: 3.141597
```

Table 9–1 summarizes the results for each of the four numerical integration methods used to approximate the area of the quarter circle with radius $r = 2$.

To further compare the two methods listed in Table 1–1, consider the half-cycle sine wave shown in Figure 9–11. The exact area under the curve from $x = 0$ to $x = \pi$ ($\pi = 3.141593$, accurate to six decimal places) can be shown using calculus to be 2.0. To approximate this value using numerical integration, Programs 9-10 through 9-13 were modified by replacing the calculated function to $f(x) = \sin(x)$.

When running the programs the bounds 0 and 3.141593 were entered. The tabulated results for the sine wave are listed in Table 9–2.

Using both Table 9–1 and Table 9–2 we can compare the four methods for the two examples, still far too few from which to draw any general conclusions, however. For the quarter circle the standard rectangular method has the poorest accuracy and the trapezoidal method the next poorest. This is the same for the half-cycle sine wave, which is coincidental and arises from the special properties of the symmetrical sine wave. For the quarter circle, the modified rectangular and Simpson's methods converge to the correct result equally fast as n increases.

TABLE 9–1 Comparison of Area Approximation for $f(x) = \sqrt{4 - x^2}$ over the Interval $x = 0$ to $x = 2$

n	Standard Rectangular Method	Midpoint Rectangular Method	Trapezoidal Method	Simpson's Method
2	3.372051	3.259367	2.732051	2.976068
10	3.304518	3.152411	3.104518	3.127008
50	3.178269	3.142466	3.138269	3.140305
1000	3.143580	3.141631	3.141580	3.141597
(Exact area = 3.141593)				

TABLE 9–2 Comparison of Area approximations for $fx = \sin(x)$ over the Interval $x = 0$ to $x = \pi$

n	Standard Rectangular Method	Midpoint Rectangular Method	Trapezoidal Method	Simpson's Method
2	1.570796	2.221442	1.570796	2.094395
10	1.983524	2.008249	1.983523	2.000109
50	1.999342	2.000329	1.999342	2.000000
1000	1.999998	2.000001	1.999998	2.000000
		(Exact area = 2.000000)		

Application: Finding Average and RMS Values

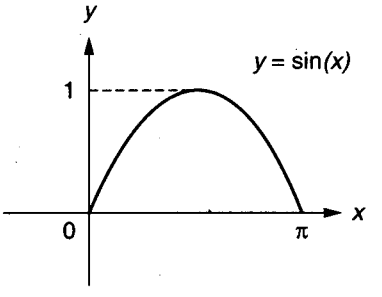
The techniques for finding the area under a curve can also be used to find the average or root-mean-squared (RMS) value of a waveform described by a function $y = f(x)$. This subject is important in electrical instrumentation where many voltage and current measuring instruments are calibrated to the average or RMS value of a periodic signal (such as a sine wave).

Refer to Figure 9–6 again, illustrating the rectangular approximation to the area under a general curve $y = f(x)$. The average value of the function can be approximated reasonably well by selecting a reasonably large number of equally spaced points on the curve, adding their y values, and dividing by the total number of points selected. Notice that this procedure is analogous to finding the average quiz grade for a class by adding the individual quiz grades and dividing by the size of the class. The average value corresponding to n points on the curve corresponding to $x_1, x_2, x_3, \ldots, x_n$ is:

$$\text{Average} = \frac{f(x_1) + f(x_2) + f(x_3) + \ldots + f(x_n)}{n}$$

$$= \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

FIGURE 9–11 Half Cycle of a Sine Wave



where the x values are separated by the fixed interval width

$$w = \frac{b-a}{n}$$

as before. The expression above can be written as

$$\begin{aligned}\text{Average} &= \left(\frac{1}{nw}\right) w \sum_{i=1}^{n} f(x_i) \\ &= \frac{1}{b-a} \; w \sum_{i=1}^{n} f(x_i) \\ &= \frac{\text{Approximate area under curve}}{\text{Total interval width}}\end{aligned}$$

This should satisfy our intuitive notion that the average height of a curve is the area divided by the base or total width.

The RMS value of a set of function values is also an average or "mean." As the name suggests, the root-mean-squared value is the square root of the average of the individual squared values. For our notation, the RMS value can be written as

$$\text{RMS} = \sqrt{\frac{f^2(x_1) + f^2(x_2) + f^2(x_3) + \ldots + f^2(x_n)}{n}}$$

Program 9-14 is a modification of the rectangular approximation technique shown in Program 9-10. The calculation of area has been replaced in the loop by the calculation of the sum of the values and the sum of the squared values of the function. After `n` iterations we exit from the loop and calculate the average by dividing `sum` by `n` and the RMS value by dividing `sumsqr` by `n` and taking the square root of the result. A sine wave is used for illustration.

Sample runs for Program 9-14 are shown below.

```
Enter the left and right bounds of x (a b): 0 3.141593
Enter the number of subintervals (n): 10
The approximate average value = 0.631375
The approximate RMS value = 0.707107

Enter the left and right bounds of x (a b): 0 3.141593
Enter the number of subintervals (n): 50
The approximate average value = 0.636410
The approximate RMS value = 0.707107

Enter the left and right bounds of x (a b): 0 6.283186
Enter the number of subintervals (n): 50
The approximate average value = 0.000000
The approximate RMS value = 0.707107
```

The first two runs approximate the average and RMS values for the half-cycle sine wave shown in Figure 9–11, from $x = 0$ to $x = \pi$ (where $\pi = 3.141593$). Recall

Program 9-14

```
#include <stdio.h>
#include <math.h>
main( )
{
  int i, n;
  float x, a, b, width, sum, sumsqr, average, rms;
  float f(float);   /* function prototype */

  printf("Enter the left and right bounds of x (a b): ");
  scanf("%f %f", &a, &b);
  printf("Enter the number of subintervals (n): ");
  scanf("%d", &n);
  width = (b - a) / n;
  x = a;
  sum = 0.0;
  sumsqr = 0.0;
  for ( i = 1; i <= n; ++i)
  {
    sum = sum + f(x);
    sumsqr = sumsqr + pow(f(x),2);
    x = x + width;
  }
  average = sum /n;
  rms = sqrt(sumsqr/n);
  printf("The approximate average value = %f", average);
  printf("\nThe approximate RMS value = %f", rms);
}
/* the function is evaluated below */
float f(float x)
{
  return( sin(x) );
}
```

that the exact area under this curve is 2.0, and since the total interval width is π, the actual average value is:

$$\text{Average} = 2 / \pi = 2 / 3.141593 = 0.636620$$

The RMS value can be derived with the aid of calculus and is found to be:

$$\text{RMS} = 1 / \sqrt{2} = 0.707107$$

Notice that the computed average is closer to the exact value for larger n, as expected. The RMS value, interestingly, is nearly exact for the smaller as well as

the larger value of n. As a result of the special properties of the sine function you will obtain nearly the exact RMS value for any n (try it for yourself!).

The final sample run shown is for a full cycle of the sine wave, from $x = 0$ to $x = 2 \cdot \pi$. The sine wave in the interval from π to $2 \cdot \pi$ has the same shape as in Figure 9–11, but flipped over to the negative direction. The sum of positive and negative values cancel one another, resulting in an average value of zero. However, the negative values when squared become positive values and the resulting RMS value is the same as for the half-cycle case (do you see why ?). The results of the last run support these conclusions.

Exercises 9.3

1 a. Modify Program 9-10 to approximate the area under the curve

$$y = x^3 + 2x^2 + 3x + 1$$

in the interval from $x = 0$ to $x = 1$. Run the program four times for $n = 2, 10, 50$, and 1000 respectively. The exact area is:

$$\text{Area} = 41/12 = 3.416666$$

b. Repeat Exercise 1a using the modified rectangular method (Program 9-11).

2 a. Repeat Exercise 1a using the trapezoidal method (Program 9-12).
b. Repeat Exercise 1a using Simpson's method (Program 9-13).
c. Compare the results of Exercises 1 and 2.

3. Modify Program 9-11 to calculate and display a table of values of area for the quarter circle used in the text examples, for $n = 1, 2, 3, \ldots, 10$. Also, tabulate for each area the percentage error between the approximate area and the exact area of $\pi = 3.141593$. Notice that the input of n from the keyboard needs to be replaced by a second loop in the program in which n is an index varying from 1 to 10. The original loop in the program should be nested inside this second loop. The output should be displayed using the heading below:

```
   n          Approximate area          Percent error
---          ----------------          -------------
```

4. The centroid is an important concept in engineering mechanics. It represents the location of the center of gravity of a body of uniform density and thickness. For a curve described by the equation $y = f(x)$ it can be shown that the centroid is located at a point (x_c, y_c) where

$$x_c = \frac{\text{Area under function } f(x)}{\text{Area under function } f(x)}$$

$$y_c = \frac{\text{Area under function } f^2(x)/2}{\text{Area under function } f(x)}$$

Notice that finding the centroid location requires the computation of the areas under three different functions, each related to $f(x)$.

a. Modify Program 9-11 to calculate and display the approximate centroid location for the function

$$y = x^2$$

in the interval from $x = 0$ to $x = 1$.

b. Compile the program from Exercise 4a and run it for $n = 5$, 10, and 50. Comment on the relative accuracy of x_c and y_c. The exact centroid location is:

$$x_c = 0.75, y_c = 0.3$$

5. Repeat Exercise 4 for the function

$$y = x^4$$

in the interval from $x = 0$ to $x = 1$. The exact centroid location, accurate to six decimal places, is $x_c = 5/6 = 0.833333$ and $y_c = 5/18 = 0.277777$.

6 *a.* Modify Program 9-14 to calculate and display the average and RMS value of the function $y = \sin(2x)$ in the interval $x = 0$ to $x = \pi = 3.141593$. Run the program for $n = 10$ and again for $n = 50$. Compare the results with those obtained from the sample runs shown of Program 9-14 for the function $y = \sin(x)$ in the same interval.

b. Repeat Exercise 6a for the function $y = e^{-x}\sin(x)$.

7. An unusual method of approximating the area under a curve can be made using a Monte Carlo simulation algorithm. To understand this algorithm, consider Figure 9–12 in which a rectangle of base $[a,b]$ is superimposed on the curve $y = f(x)$ such that the height of the rectangle, H, is larger than or equal to any y value on the curve.

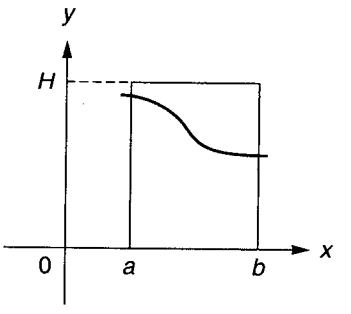
Now consider throwing N darts at Figure 9–12 and calculating the total number of darts, M, that land in the shaded region. Probability theory states that for a large number of darts the ratio of the M darts that land in the shaded area to the total number of N darts thrown is the same as the ratio of the shaded area to the total area. That is,

$$\frac{M}{N} = \frac{\text{Shaded area}}{\text{Rectangular area}}$$

Solving this formula for the shaded area under the curve yields:

$$\text{Shaded area} = (M/N) * \text{Rectangular area}$$

FIGURE 9–12



Using this information write a C program to approximate the area under the curve

$$y = \sqrt{4 - x^2}$$

from $x = 0$ to $x = 1$. Generate random values of x and y between 0 and 1 using the random number generation algorithm presented in Section 7.3. If the resultant point (x,y) lies below the curve, the generated point is considered landing in the shaded region, else it is considered landing outside of the shaded region. Run your program using 2, 10, 50, and 1000 random points, respectively, and compare the results obtained with those presented in Table 9–1.

9.4 Common Programming Errors

The common errors associated with the techniques presented in this chapter include the following. First, although a floating point variable is often used for convenience in root-finding techniques, as it is in Programs 9-5 and 9-6, this practice can lead to problems in determining the exact number of iterations performed. Specifically, because of precision errors, the `for` loop may terminate earlier or later than expected. For example, the loop

```
float x;
for (x = 0.1; x <= 10.0; x = x + 0.1)
```

may exit after `x = 9.9` rather than `x = 10.0` as intended. The reason is that after the last increment of the index its value may round off to `9.999999` or possibly to `10.000001`. In the latter case the loop is exited because the index value exceeds the upper limit of `10.0`. This problem can be avoided, when necessary, by using an integer variable as a loop counter. For example, using an integer loop counter the following routine can be used to replace the previous loop:

```
float x;
int i;
x = 0.1;
for (i = 1; i <= 100; ++i)
{
      :
      :
      :
x = x + 0.1;
}
```

The cost to this solution is the additional complexity of the resultant code.

Second, unless numerical iteration algorithms are carefully programmed, their execution times can be excessive. Both root-finding and numerical integration programs containing only single loops may exhibit a noticeable delay in completion when the number of iterations becomes larger than 50. For such programs it is frequently possible to decrease the run time by performing a calcula-

tion outside the loop rather than inside. For example, consider the following segment of code from Program 9-8:

```
x = a;
area = 0;
for ( i = 1; i <= n; ++i)
{
  area = area + width * f(x + width/2.0);
  x = x + width;
}
```

Recall that in this program x represents the start of each interval with the midpoint at `x + width / 2.0`. Within the loop the function is repeatedly evaluated at the interval midpoint, requiring `width/2.0` to be repeatedly added to x. By initializing x to the midpoint of the first interval, however, we can move this calculation outside the loop. Similarly, the returned value of the function `f(x + width/2.0)` is multiplied by `width` within the existing loop before being added to the previously stored value of `area`. Since `width` has a constant value this multiplication can be done after the loop is exited. The previous segment of code, therefore, can be replaced by

```
x = a + width/2.0;
area = 0;
for ( i = 1; i <= n; ++i)
{
  area = area + f(x);
  x = x + width;
}
area = width * area;
```

Since the updating of `area` inside the loop now involves fewer calculations, the program will run faster. For the sake of readability you may want to use a name other than `area` inside the loop, since the value returned when the loop is exited is not the actual "area" (until it is multiplied by the subinterval width).

9.5 Chapter Summary

1. The solution of linear equations with two or three unknowns can be easily programmed. One such method uses *Cramer's Rule* and is conveniently programmed using a function for the calculation of the required determinants. This algorithm is not efficient, however, for larger numbers of equations and unknowns.
2. There are several programming techniques for finding the real roots of an equation $y = f(x)$. In *fixed-increment* techniques the value of x is incremented by a constant step size over a range of values that includes the roots of interest. The value of y is calculated for each x and a root is assumed to exist wherever the absolute value of y is arbitrarily small. Alternatively, the roots can be located where the function changes sign for two successive values of x.

Although fixed-increment methods are easy to program, particularly with the aid of `for` loops, such programs generally require a relatively long time to run.

3. The *bisection method* for root finding locates roots more rapidly than do fixed-increment methods. Using a bisection algorithm the interval of x is repeatedly bisected. For each bisection the half-interval that maintains the sign change at the endpoints is retained and the other half-interval is discarded from further consideration. The process continues until the interval is arbitrarily small. The presence of multiple roots can cause problems with this method.
4. The *secant method* for root finding is usually more efficient than the bisection method. Rather than bisecting the interval, a secant line is drawn from the points on the function's curve corresponding to the endpoints of the interval. The intersection of the secant line with the x-axis is then used as one of the endpoints of the next interval. The process terminates when the absolute value of y is arbitrarily small. Like the bisection method, the secant method may have difficulties handling multiple roots.
5. Numerical integration is a technique for approximating the area under an interval of a curve described by a function $y = f(x)$. The most commonly used techniques divide the interval into n subintervals, where n is an arbitrary number, and adds the subinterval areas in sequence. Within each subinterval the curve can be approximated by a constant value, in which case the total area is the sum of rectangular areas. Alternatively, a straight line can be connected between the endpoints of the subinterval, leading to the addition of trapezoidal areas. *Simpson's method* involves approximating the function in two successive subintervals using a parabolic curve. All of the approximation techniques can be applied to finding the average and root-mean-square (RMS) values of functions.

Pointers

Chapter Ten

10.1 Addresses and Pointers
10.2 Returning Multiple Function Values
10.3 Array Names as Pointers
10.4 Pointer Arithmetic
10.5 Passing and Using Array Addresses
10.6 Passing and Returning File Names
10.7 Common Programming Errors
10.8 Chapter Summary

One of C's advantages is that it allows the programmer access to the addresses of variables used in a program. Although we have already used the address operator, &, in calling the `scanf( )` function, the real power of using addresses is that it allows a programmer to enter directly into the computer's inner workings and access the computer's basic storage structure. This gives the C programmer capabilities and programming power that is not available in other high-level languages.

This chapter presents the basics of declaring variables to store addresses. Such variables are referred to as pointer variables, or simply pointers. Additionally, we discuss methods of using pointer variables to access and use their stored addresses in meaningful ways.

10.1 Addresses and Pointers

Every variable has three major items associated with it: its data type, the actual value stored in the variable, and the address of the variable. In C, a variable's data type is declared using a declaration statement. The actual value stored in a variable is formally referred to as the variable's *rvalue*, and the address of the variable (where the variable is located in memory) is referred to as the variable's *lvalue*. These two terms make more sense when a typical illustration of a variable is considered, as shown in Figure 10–1. As seen in this figure, the address of the variable is typically written on the left of the figure and the variable's contents (the value in the box) on the right.

Programmers are usually concerned only with the value assigned to a variable (its contents, or rvalue) and give little attention to where the value is stored (its address, or lvalue). For example, consider Program 10-1.

Program 10-1

```
#include <stdio.h>
main( )
{
  int num;

  num = 22;
  printf("The value stored in num is %d.",num);
  printf("\nThe computer uses %d bytes to store this value",sizeof(num));

}
```

The output displayed when Program 10-1 is run is:

```
The value stored in num is 22
The computer uses 2 bytes to store this value
```

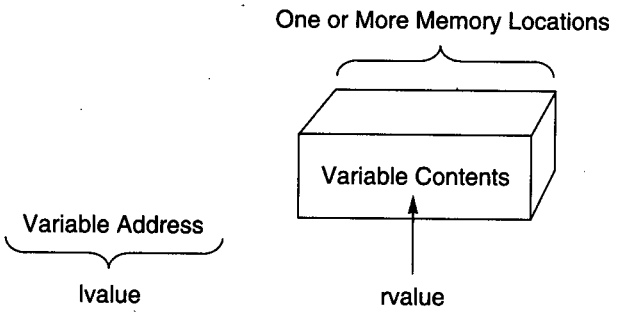


FIGURE 10–1 A Typical Variable

Program 10-1 displays both the number 22, which is the value stored in the integer variable num (its rvalue), and the amount of storage used for this number. The information provided by Program 10-1 is illustrated in Figure 10–2.

We can go further and obtain the address, or lvalue, corresponding to the variable num. The address that is displayed corresponds to the address of the first byte set aside in the computer's memory for the variable.[1]

To determine the address of num, we must use the address operator, &, which means *the address of*, directly in front of the variable name (no space between & and the variable). For example, &num means the address of num, &total means the address of total, and &price means the address of price. Program 10-2 uses the address operator to display the address of the variable num.

Program 10-2

```
#include <stdio.h>
main( )
{
  int num;

  num = 22;
  printf("num = %d  The address of num = %p", num, &num);
}
```

The output of Program 10-2 is:

```
num = 22  The address of num = FFE0
```

Figure 10–3 illustrates the additional address information provided by the output of Program 10-2.

Clearly, the address output by Program 10-2 depends on the computer used to run the program. Every time Program 10-2 is executed, however, it displays

[1] Review section 1.7 if you are unfamiliar with the concept of memory addresses and bytes.

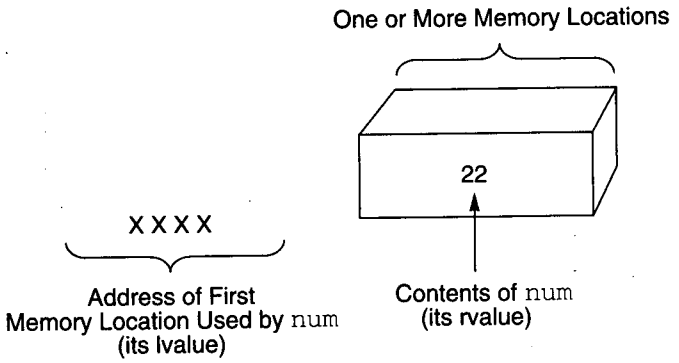


FIGURE 10–2 Somewhere in Memory

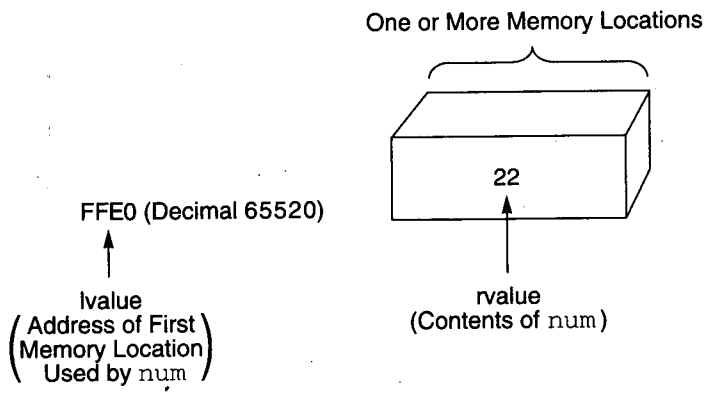
the address of the first byte used to store the variable num. Note also that the address is printed using the conversion sequence %p. This conversion sequence is provided in ANSI C to display addresses.[2] As illustrated by the output of Program 10-2, the display is in hexadecimal notation. This display has no effect on how addresses are used internal to the program; it merely provides us with a means of displaying addresses that is helpful in understanding them. As we shall see, using addresses as opposed to only displaying them provides the C programmer with an extremely powerful programming tool.

Storing Addresses

Besides displaying the address of a variable, as was done in Program 10-2, we can also store addresses in suitably declared variables. For example, the statement

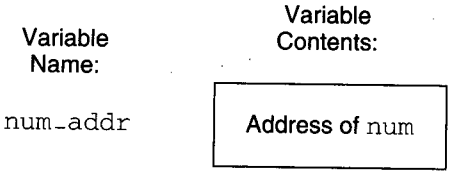
```
num_addr = &num;
```

FIGURE 10–3 A More Complete Picture of the Variable num



[2] For non-ANSI compilers that do not provide the %p conversion sequence, the unsigned conversion sequence %u may be used in its place. The %u conversion sequence forces the address to be treated as an unsigned integer data type, and what is displayed is printf( )'s representation of the address in a decimal format. An address, however, is not an unsigned integer data type—it is a unique data type that may or may not require the same amount of storage as an unsigned integer.

FIGURE 10–4 Storing num's Address into num_addr



stores the address corresponding to the variable num in the variable num_addr, as illustrated in Figure 10–4. Similarly, the statements

```
d = &m;

tab_point = &list;

chr_point = &ch;
```

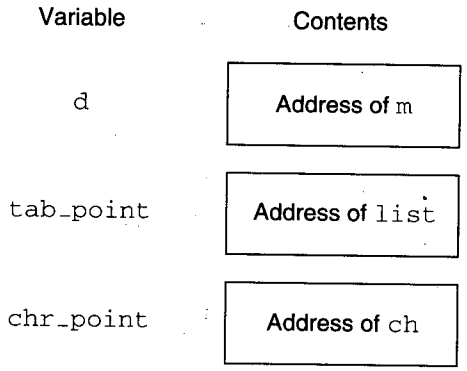
store the addresses of the variables m, list, and ch in the variables d, tab_point, and chr_point, respectively, as illustrated in Figure 10–5.

The variables num_addr, d, tab_point, and chr_point are formally called pointer variables, or pointers for short. *Pointers* are simply variables that are used to store the addresses of other variables.

Using Addresses

To use a stored address, C provides us with an indirection operator, *. The * symbol, when followed immediately by a pointer (no space allowed between the * and the pointer), means "the variable whose address is stored in." Thus, if num_addr is a pointer (remember that a pointer is a variable that contains an address), *num_addr means the variable whose address is stored in num_addr. Similarly, *tab_point means the variable whose address is stored in tab_point and *chr_point means the variable whose address is stored in chr_point. Figure 10–6 shows the relationship between the address contained in a pointer variable and the variable ultimately addressed.

FIGURE 10–5 Storing More Addresses



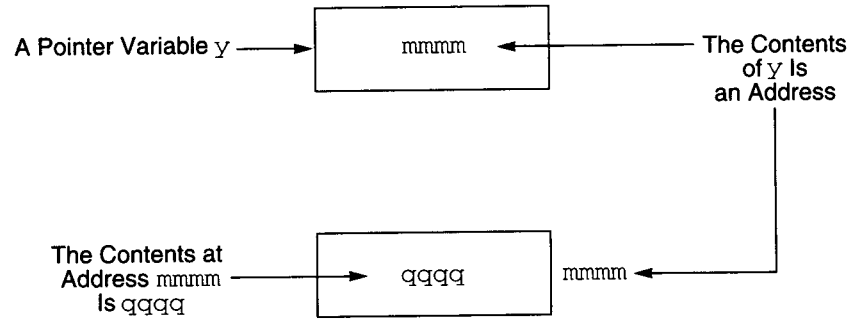


FIGURE 10–6 Using a Pointer Variable

Although `*d` literally means "the variable whose address is stored in `d`," this is commonly shortened to the statement "the variable pointed to by `d`." Similarly, referring to Figure 10–6, `*y` can be read as the variable pointed to by `y`. The value ultimately obtained, as shown in Figure 10–6, is `qqqq`.

When using a pointer variable, the value that is finally obtained is always found by first going to the pointer variable (or pointer, for short) for an address. The address contained in the pointer is then used to get the desired contents. Certainly, this is a rather indirect way of getting to the final value and, not unexpectedly, the term *indirect addressing* is used to describe this procedure.

Since using a pointer requires the computer to do a double lookup (first the address is retrieved, then the address is used to retrieve the actual data), a worthwhile question is, why would you want to store an address in the first place? The answer to this question must be deferred until we get to real applications, when the use of pointers becomes invaluable. However, given what was previously presented for a variable's storage locations, the idea of storing an address should not seem strange.

Declaring Pointers

Like all variables, pointers must be declared before they can be used. When we declare a pointer variable, C requires that we also specify the type of variable that is pointed to. For example, if the address in the pointer `num_addr` is the address of an integer, the correct declaration for the pointer is:

```
int *num_addr;
```

This declaration is read as "the variable pointed to by `num_addr` (from the `*num_addr` in the declaration) is an integer."

Notice that the declaration `int *num_addr;` specifies two things: first, that the variable pointed to by `num_addr` is an integer; second, that `num_addr` must be a pointer (because it is used with the indirection operator `*`). Similarly, if the pointer `tab_point` points to (contains the address of) a floating point number and `chr_point` points to a character variable, the required declarations for these pointers are:

```
float *tab_point;

char *chr_point;
```

These two declarations can be read, respectively, as the variable pointed to by `tab_point` is a `float` and the variable pointed to by `chr_point` is a `char`. Notice that in declaring pointers, the data type corresponding to the address being stored must be included in the declaration. Since all addresses appear the same, this additional information is needed by the computer to know how many storage locations to access when it uses the address stored in the pointer. Further examples of pointer declarations are:

```
char *in_addr;

int *num_pt;

float *dst_addr:

double *nm1_addr;
```

To understand pointer declarations, it is helpful to read them "backwards," starting with the indirection operator, the asterisk, *, and translating it either as "the variable whose address is stored in" or "the variable pointed to by." Applying this to pointer declarations, the declaration `char *in_key;`, for example, can be read as either "the variable whose address is stored in `in_key` is a character" or "the variable pointed to by `in_key` is a character." Both of these statements are frequently shortened to the simpler statement "`in_key` points to a character." As all three interpretations of the declaration statement are correct, select and use whichever description makes a pointer declaration meaningful to you. We now put this together to construct a program using pointers. Consider Program 10-3.

Program 10-3

```
#include <stdio.h>
main( )
{
  int *num_addr;        /* declare a pointer to an int       */
  int miles, dist;      /* declare two integer variables     */

  dist = 158;           /* store the number 158 into dist    */
  miles = 22;           /* store the number 22 into miles    */
  num_addr = &miles;    /* store the 'address of miles' in num_addr*/

  printf("The address stored in num_addr is %p\n",num_addr);
  printf("The value pointed to by num_addr is %d\n\n",*num_addr);

  num_addr = &dist;      /* now store the address of dist in num_addr*/
  printf("The address now stored in num_addr is %p\n",num_addr);
  printf("The value now pointed to by num_addr is %d\n",*num_addr);

}
```

The output of Program 10-3 is:

```
The address stored in num_addr is FFE0
The value pointed to by num_addr is 22

The address now stored in num_addr is FFE2
The value now pointed to by num_addr is 158
```

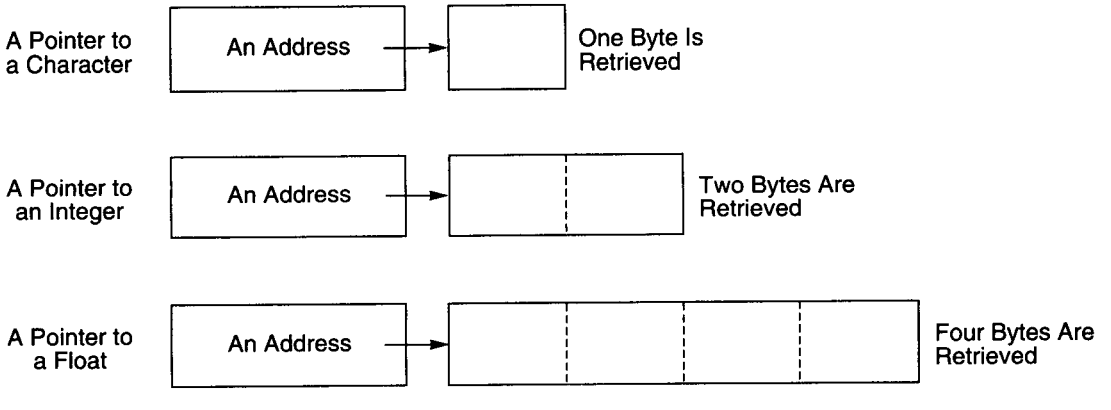
The only value of Program 10-3 is in helping us understand "what gets stored where." Let's review the program to see how the output was produced.

The declaration statement `int *num_addr;` declares `num_addr` to be a pointer variable used to store the address of an integer variable. The statement `num_addr = &miles;` stores the address of the variable `miles` into the pointer `num_addr`. The first call to `printf( )` causes this address to be displayed. Notice that we have again used the control sequence `%p` to print out the address. The second call to `printf( )` in Program 10-3 uses the indirection operator to retrieve and print out the value pointed to by `num_addr`, which is, of course, the value stored in `miles`.

Since `num_addr` has been declared as a pointer to an integer variable, we can use this pointer to store the address of any integer variable. The statement `num_addr = &dist;` illustrates this by storing the address of the variable `dist` in `num_addr`. The last two `printf( )` calls verify the change in `num_addr`'s value and that the new stored address does point to the variable `dist`. As illustrated in Program 10-3, only addresses should be stored in pointers.

It certainly would have been much simpler if the pointer used in Program 10-3 could have been declared as `point num_addr;`. Such a declaration, however, conveys no information as to the storage used by the variable whose address is stored in `num_addr`. This information is essential when the pointer is used with the indirection operator, as it is in the second `printf( )` call in Program 10-3. For example, if the address of an integer is stored in `num_addr`, then only two bytes of storage are typically retrieved when the address is used. If the address of a character is stored in `num_addr`, only one byte of storage would be retrieved, and a float typically requires the retrieval of four bytes of storage. The declaration of a pointer must, therefore, include the type of variable being pointed to. Figure 10–7 illustrates this concept.

FIGURE 10–7 Addressing Different Data Types Using Pointers



Exercises 10.1

1. If `average` is a variable, what does `&average` mean?

2. For the variables and addresses illustrated in Figure 10–8, determine `&temp`, `&dist`, `&date`, and `&miles`.

3a. Write a program that includes the following declaration statements. Have the program use the address operator and the `printf( )` function to display the addresses corresponding to each variable.

```
char key, choice;
int num, count;
long date;
float yield;
double price;
```

b. After running the program written for Exercise 3a, draw a diagram of how your computer has set aside storage for the variables in the program. On your diagram, fill in the addresses displayed by the program.

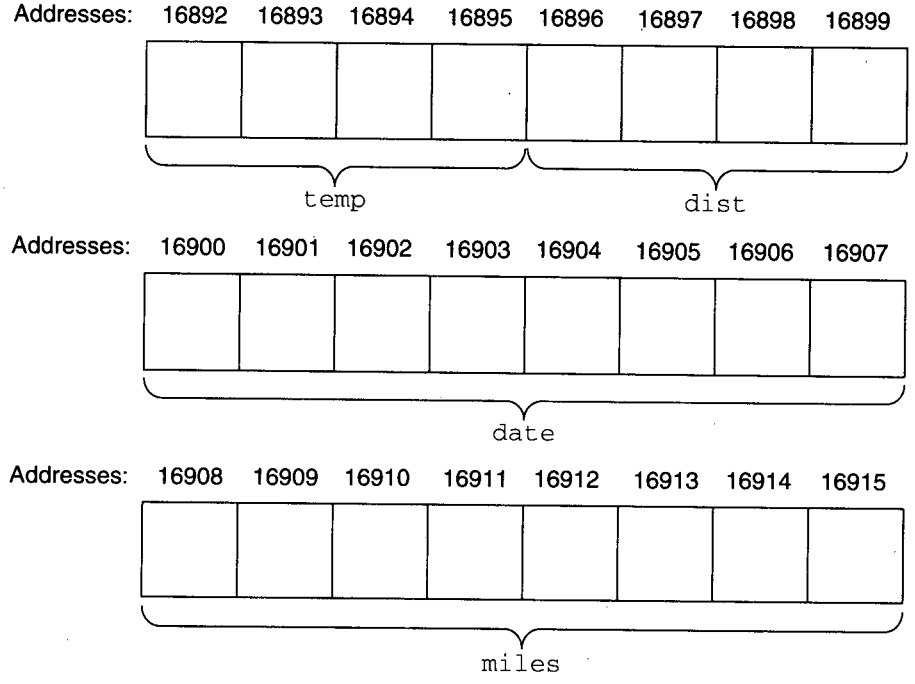
c. Modify the program written in Exercise 3a to display the amount of storage your computer reserves for each data type (use the `sizeof( )` operator). With this information and the address information provided in Exercise 3b, determine if your computer set aside storage for the variables in the order they were declared.

4. If a variable is declared as a pointer, what must be stored in the variable?

5. Using the indirection operator, write expressions for the following:

a. The variable pointed to by `x_addr`

FIGURE 10–8 Memory Bytes for Exercise 2



b. The variable whose address is in `y_addr`
c. The variable pointed to by `pt_yld`
d. The variable pointed to by `pt_miles`
e. The variable pointed to by `mptr`
f. The variable whose address is in `pdate`
g. The variable pointed to by `dist_ptr`
h. The variable pointed to by `tab_pt`
i. The variable whose address is in `hours_pt`

6. Write declaration statements for the following:
a. The variable pointed to by `y_addr` is an integer.
b. The variable pointed to by `ch_addr` is a character.
c. The variable pointed to by `pt_yr` is a long integer.
d. The variable pointed to by `amt` is a double precision variable.
e. The variable pointed to by `z` is an integer.
f. The variable pointed to by `qp` is a floating point variable.
g. `date_pt` is a pointer to an integer.
h. `yld_addr` is a pointer to a double precision variable.
i. `amt_pt` is a pointer to a floating point variable.
j. `pt_chr` is a pointer to a character.

7a. What are the variables `y_addr`, `ch_addr`, `pt_yr`, `amt`, `z`, `qp`, `date_ptr`, `yld_addr`, `amt_pt`, and `pt_chr` used in Exercise 6 called?
b. Why are the variable names `amt`, `z`, and `qp` used in Exercise 6 not good choices for pointer variable names?

8. Write English sentences that describe what is contained in the following declared variables:
a. `char *key_addr;`
b. `int *m;`
c. `double *yld_addr;`
d. `long *y_ptr;`
e. `float *p_cou;`
f. `int *pt_date;`

9. Which of the following are declarations for pointers?
a. `long a;`
b. `char b;`
c. `char *c;`
d. `int x;`
e. `int *p;`
f. `double w;`
g. `float *k;`
h. `float l;`
i. `double *z;`

10. For the following declarations,

```
int *x_pt, *y_addr;
long *dt_addr;
double *pt_z;
int a;
long b;
double c;
```

determine which of the following statements is valid.

a. `y_addr = &a;` *b.* `y_addr = &b;` *c.* `y_addr = &c;`
d. `y_addr = a;` *e.* `y_addr = b;` *f.* `y_addr = c;`
g. `dt_addr = &a;` *h.* `dt_addr = &b;` *i.* `dt_addr = &c;`

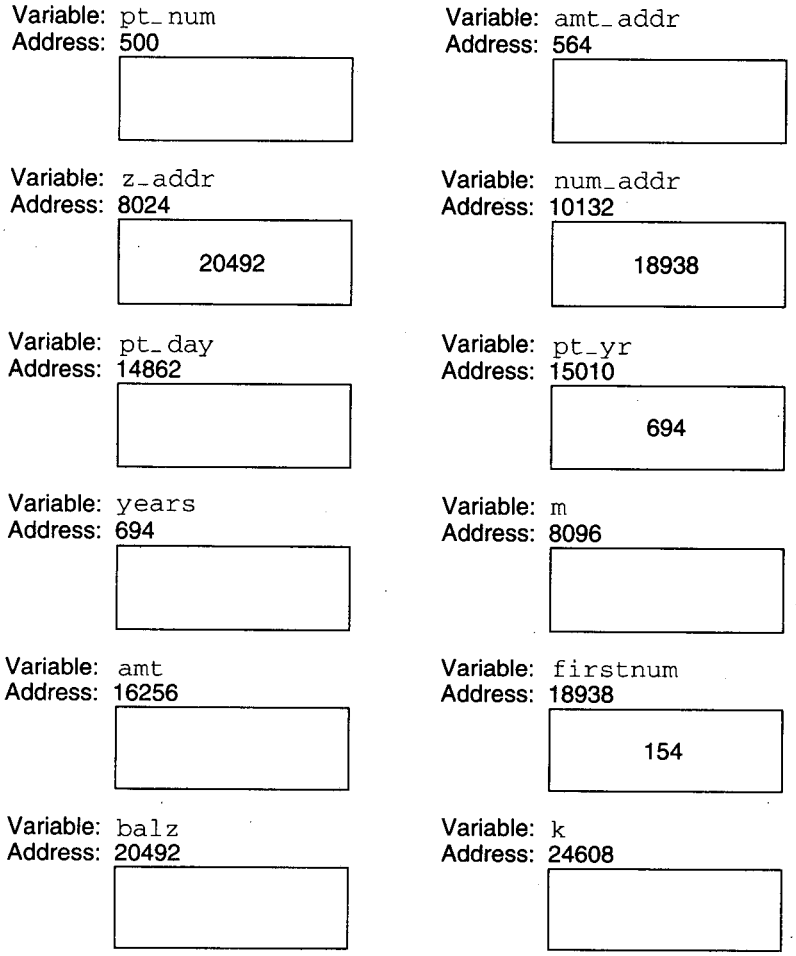
j. `dt_addr = a;` *k.* `dt_addr = b;` *l.* `dt_addr = c;`
m. `pt_z = &a;` *n.* `pt_addr = &b;` *o.* `pt_addr = &c;`
p. `pt_addr = a;` *q.* `pt_addr = b;` *r.* `pt_addr = c;`
s. `y_addr = x_pt;` *t.* `y_addr = dt_addr;` *u.* `y_addr = pt_addr;`

11. For the variables and addresses illustrated in Figure 10–9, fill in the appropriate data as determined by the following statements:

a. `pt_mon = &m;`
b. `amt_addr = &amt;`
c. `*z_addr = 25;`
d. `k = *num_addr;`
e. `pt_day = z_addr;`
f. `*pt_yr = 1987;`
g. `*amt_addr = *num_addr;`

12. Using the `sizeof( )` operator, determine the number of bytes used by your computer to store the address of an integer, character, and double precision number. (*Hint:* `sizeof (*int)` can be used to determine the number of memory bytes used for a pointer to an integer.) Would you expect the size of each address to be the same? Why or why not?

FIGURE 10–9 Memory Locations for Exercise 11



10.2 Returning Multiple Function Values

As we saw in Chapter 7, a called function receives values from its calling function, stores the passed values in its own local arguments, manipulates these arguments appropriately, and possibly returns a single value. Under this standard arrangement the called function does not receive access to any variables used in the call; it only receives copies of the values stored in the variables. For example, in the function call

```
max = find_max(firstnum, secnum);
```

the `find_max` function receives copies of the values contained in `firstnum` and `secnum` when the call is made. This method of calling a function and passing values to it is referred to as a function *call by value*.

This call by value procedure is a distinct advantage of C. It allows functions to be written as independent entities that can use any local argument and variable names without concern that other functions may also be using the same names. When we write a function, we can conveniently think of arguments as variables that will be assigned values when the function is executed.

There are times, however, when it is desirable to give a function access to the local variables of its calling function. This allows the called function to directly access and change values in the variables of the called function, and effectively permits the called function to return multiple values. Doing this requires that a variable's address be passed to the called function. Once the called function has the variable's address, it "knows where the variable lives," so to speak, and can access the variable using the address and the indirection operator.

Passing addresses is referred to as a function *call by reference*, since the called function can reference, or access, the variable using the passed address. For example, the function call

```
sort_num(&firstnum, &secnum);
```

passes the addresses of the variables `firstnum` and `secnum` to `sortnum( )`. If `sortnum( )` has been written to correctly receive these addresses, it can use them to directly alter the contents of `firstnum` and `secnum`. Addresses, as we saw in the previous section, are stored in pointers, which means that the arguments of the `sortnum( )` function must be declared as pointers. The `sortnum( )` function will be written to compare the values contained in the passed addresses and swap the values, if necessary, so that the smaller value is stored in the first address.

Passing addresses to a function should be familiar to you, because we have been using addresses each time we called the `scanf( )` function. Consider Program 10-4.

Observe in Program 10-4 that the calls to both `scanf( )` and `sortnum( )` involve passing the addresses of variables (see Figure 10–10). Passing addresses to `sortnum( )` permits the function to directly access the variables `firstnum` and `secnum` because it knows where these variables are located in the computer's memory. Notice that had just the values of `firstnum` and `secnum` been passed using the call `sortnum(firstnum, secnum);` the function could not

Program 10-4

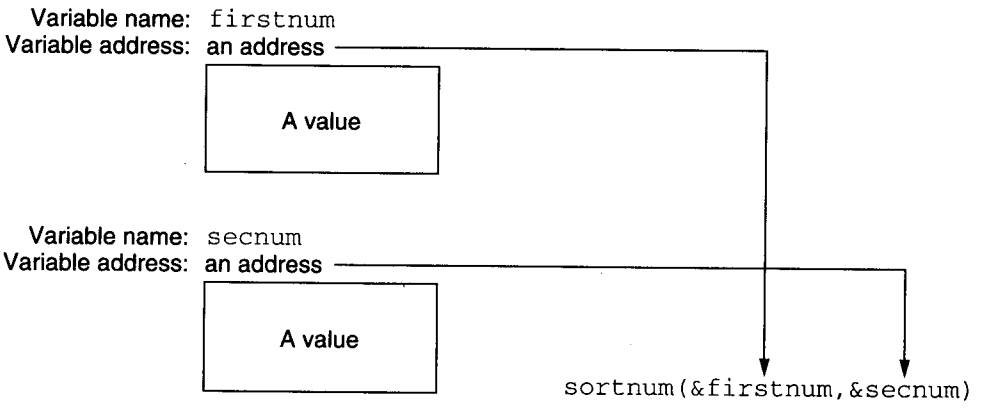
```
#include <stdio.h>
main( )
{
  double firstnum, secnum;
  void sortnum(double *, double *);     /* function prototype */

  printf("Enter two numbers: ");
  scanf("%lf %lf", &firstnum, &secnum);

  sortnum(&firstnum, &secnum);

  printf("\nThe smaller number is %lf", firstnum);
  printf("\nThe larger number is %lf",secnum);
}
```

be written to swap the values within the two variables. This is because `sortnum( )` would have no way of determining where the values came from. The function prototype for `sortnum( )` declares that the function does not directly return a value and expects two pointers, each of which points to (is the address of) a double precision value.

One of the first requirements in writing `sortnum( )` is to write a program header line that correctly declares the data types of the function's arguments. In this case, `sortnum( )` is required to receive and store two addresses, each of which is the address of a double precision value. A suitable header line for `sortnum( )`, therefore, is:

```
void sortnum(double *nm1_addr, double *nm2_addr)
```

FIGURE 10–10 Passing Addresses to `sortnum( )`



The choice of the argument names nm1_addr and nm2_addr is, as with all argument names, up to the programmer. The declaration double *num1_addr, however, declares that the argument named num1_addr will be used to store an address of a double precision value. Similarly, the declaration double *num2_addr declares that num2_addr will also store the address of a double precision value.

Before writing the body of sortnum( ) to actually compare and swap values, let's first check that the values accessed using the addresses in nm1_addr and nm2_addr are correct. This is done in Program 10-5.

Program 10-5

```
#include <stdio.h>
main( )
{
  double firstnum = 20.0, secnum = 5.0;
  void sortnum(double *, double *);  /* function prototype */

  sortnum(&firstnum, &secnum);
}

void sortnum(double *nm1_addr, double *nm2_addr)
{
  printf("The number whose address is in nm1_addr is %lf", *nm1_addr);
  printf("\nThe number whose address is in nm2_addr is %lf", *nm2_addr);

}
```

The output displayed when Program 10-5 is run is:

```
The number whose address is in nm1_addr is 20.000000
The number whose address is in nm2_addr is  5.000000
```

In reviewing Program 10-5, note two things. First, the function prototype for sortnum( )

```
void sortnum(double *, double *)
```

declares that sortnum( ) returns no value directly and that its arguments are two pointers that "point to" double precision values. As such, when the function is called it will require that two addresses be passed, and that each address is the address of a double precision value.

The second item to notice is that within sortnum( ) the indirection operator is used to access the values stored in firstnum and secnum. sortnum( ) itself has no knowledge of these variable names, but it does have the address of

firstnum stored in nm1_addr and the address of secnum stored in nm2_addr. The expression *nm1_addr used in the first printf( ) call means "the variable whose address is in nm1_addr." This is of course the variable firstnum. Similarly, the second printf( ) call obtains the value stored in secnum as "the variable whose address is in nm2_addr." Thus, we have successfully used pointers to allow sortnum( ) to access variables in main( ). Figure 10–11 illustrates the concept of storing addresses in pointer arguments.

Having verified that sortnum( ) can access main( )'s local variables firstnum and secnum, we can now expand sortnum( ) to compare the values in these variables with an if statement to see if they are in the desired order. Using pointers, the if statement takes the form

```
if (*nm1_addr > *nm2_addr)
```

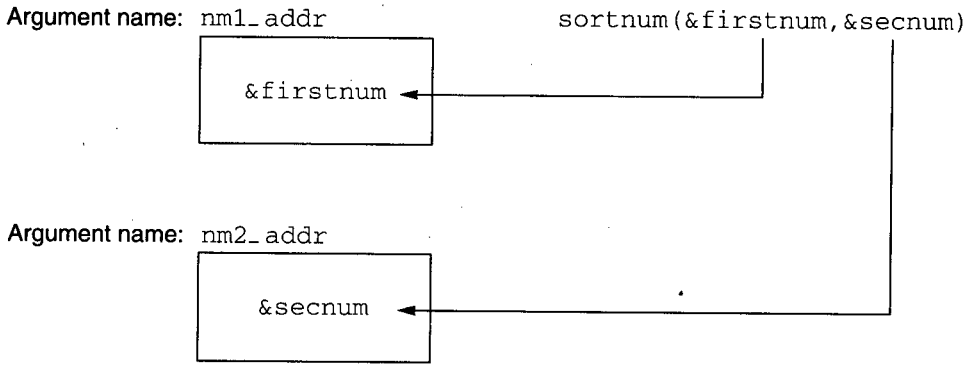
This statement should be read: "if the variable whose address is in nm1_addr is larger than the variable whose address is in nm2_addr." If the condition is true, the values in main( )'s variables firstnum and secnum can be interchanged from within sortnum( ) using this three-step interchange algorithm:

1. Store firstnum's value in a temporary location.
2. Store secnum's value in firstnum.
3. Store the temporary value in secnum.

Using pointers from within sortnum( ), this takes the form:

1. Store the variable pointed to by nm1_addr in a temporary location. The statement temp = *nm1_addr; does this (see Figure 10–12).
2. Store the variable whose address is in nm2_addr in the variable whose address is in nm1_addr. The statement *nm1_addr = *nm2_addr; does this (see Figure 10–13).
3. Move the value in the temporary location into the variable whose address is in nm2_addr. The statement *nm2_addr = temp; does this (see Figure 10–14).

FIGURE 10–11 Storing Addresses in pointer Arguments



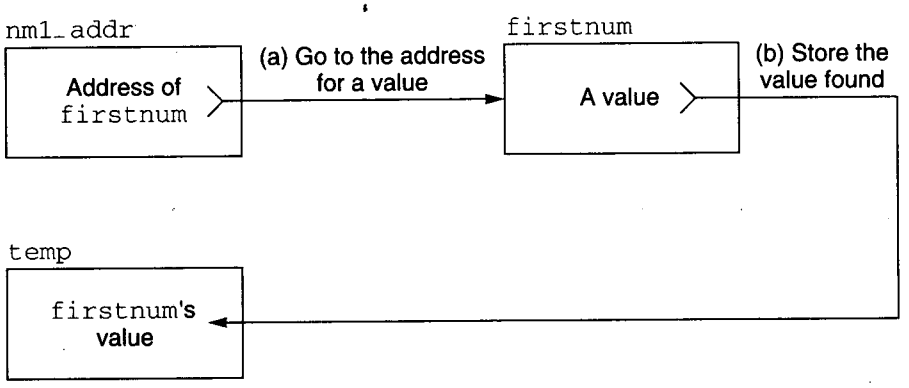


FIGURE 10–12 Indirectly Storing `firstnum`'s value

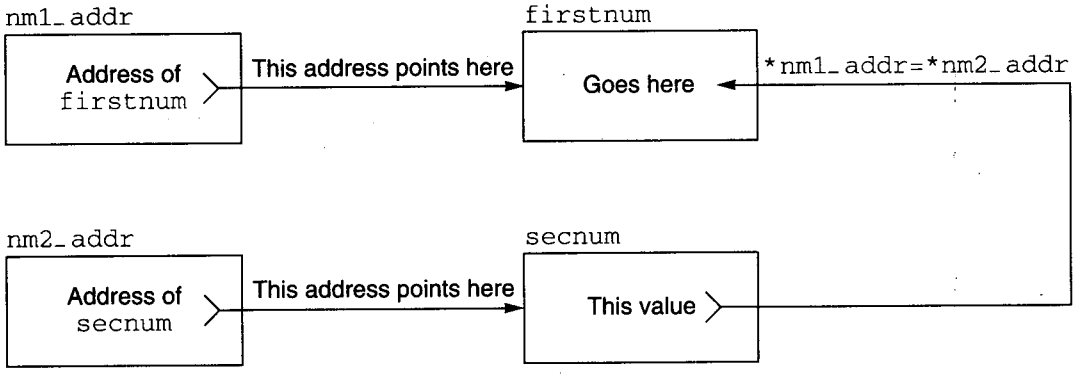


FIGURE 10–13 Indirectly Changing `firstnum( )`'s Value

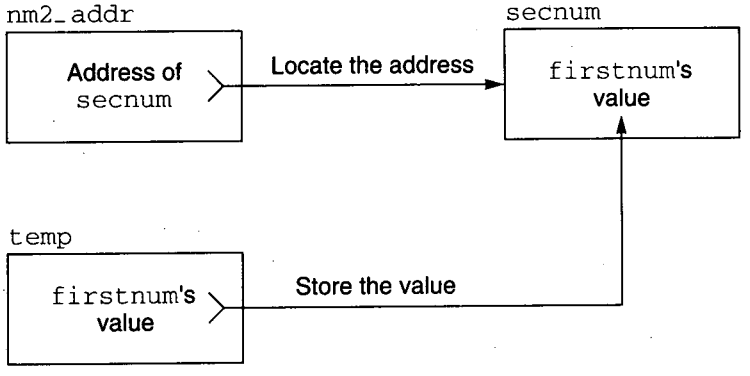


FIGURE 10–14 Indirectly Changing `secnum`'s Value

Program 10-6 contains the final form of `sortnum( )`, written according to our description.

Program 10-6

```
#include <stdio.h>
main( )
{
  double firstnum, secnum;
  void sortnum(double *, double *);  /* function prototype */

  printf("Enter two numbers: ");
  scanf("%lf %lf", &firstnum, &secnum);
  sortnum(&firstnum, &secnum);              /* call sortnum( ) */
  printf("The smaller number is %6.2lf", firstnum);
  printf("\nThe larger number is %6.2lf",secnum);
}

void sortnum(double *nm1_addr, double *nm2_addr)
{
  double temp;

  if (*nm1_addr > *nm2_addr)
  {
    temp = *nm1_addr;        /* save firstnum's value */
    *nm1_addr = *nm2_addr;   /* move secnum's value in firstnum */
    *nm2_addr = temp;        /* change secnum's value */
  }
  return;
}
```

The following sample run was obtained using Program 10-6:

```
Enter two numbers: 20.6 3.9
The smaller number is  3.90
The larger number is  20.60
```

In reviewing Program 10–6, note that the sole reason for using pointers is to allow us to swap `firstnum`'s value within `sortnum( )` if the values are not in the desired order.

Exercises 10.2

1. Write declarations for:

   a. a formal argument named `price` that will be used to accept the address of a double precision number

   b. a formal argument named `minutes` that will be used to accept the address of an integer number

   c. a formal argument named `key` that will be used to accept the address of a character

d. a formal argument named `yield` that will be used to store the address of a double precision number

2. The addresses of the integer variables `sec`, `min`, and `hours` are to be passed to a function named `time(a,b,c)`. Write a suitable function header for `time( )`.

3. Rewrite the `sortnum( )` function in Program 10-6 so that a variable `max` is declared in `main( )` and the maximum value of the two passed numbers is written directly to `max`. *Hint:* The address of `max` will also have to be passed to `sortnum( )`.

4. Write a function named `time( )` that accepts an integer number of seconds and the addresses of three variables named `hours`, `min`, and `sec`. The function is to convert the passed number of seconds into an equivalent number of hours, minutes, and seconds and directly alter the value of the respective variables using their passed addresses.

5a. The time in hours, minutes, and seconds is to be passed to a function named `secs( )`. Write `secs( )` to accept these values and determine the total number of seconds in the passed data. Write this function so that the total number of seconds is returned by the function as an integer number.

b. Repeat Exercise 5a but also pass the address of the variable `tot_sec` to the function `secs( )`. Using this passed address, have `secs( )` directly alter the value of `tot_sec`.

6. Write a function named `change( )` that accepts a floating point number and the addresses of the integer variables named `quarters`, `dimes`, `mickels`, and `pennies`. The function should determine the number of quarters, dimes, nickels, and pennies in the number passed to it and write these values directly into the respective variables declared in its calling function.

7. Write a function named `yr_calc( )` that accepts a long integer representing the total number of days from the turn of the century and the addresses of the variables `year`, `month`, and `day`. The function is to calculate the current year, month, and day for the given number of days and write these values directly in the respective variables using the passed addresses. For this problem assume that each year has 360 days and each month has 30 days.

8. Write a function named `liquid( )` that is to accept an integer number and the addresses of the variables `gallons`, `quarts`, `pints`, and `cups`. The passed integer represents the total number of cups and the function is to determine the number of gallons, quarts, pints, and cups in the passed value. Using the passed addresses, the function should directly alter the respective variables in the calling function. Use the relationships of two cups to a pint, four cups to a quart, and 16 cups to a gallon.

9. The following program uses the same variable names in both the calling and called function. Determine if this causes any problem for the computer. What problems might this code pose to a programmer? Also determine the type of data stored in each variable.

```
#include <stdio.h>
main( )
{
  int min, hour;

  printf("Enter two numbers :");
  scanf("%d %d", &min, &hour);
  time(&min,&hour);
}

time(int *min, int *hour)
{
  int sec;

  sec = ( (*hour) * 60 + *min ) * 60;
  printf("The total number of seconds is %d", sec);
}
```

10. Assume that the following declaration has been made:

```
int *pt1, *pt2;
```

Since the asterisk, *, is used for both multiplication and indirection, how is the expression

```
* pt1 * * pt2
```

evaluated by the computer? Why does the computer evaluate the expression in the order you have indicated? Rewrite this expression to make its meaning clearer to anyone reading it.

10.3 Array Names as Pointers

Although pointers are simply, by definition, variables used to store addresses, there is also a direct and intimate relationship between array names and pointers. In this section we describe this relationship in detail.

Figure 10–15 illustrates the storage of a single-dimensional array named `grades`, which contains five integers. Assume that each integer requires two bytes of storage.

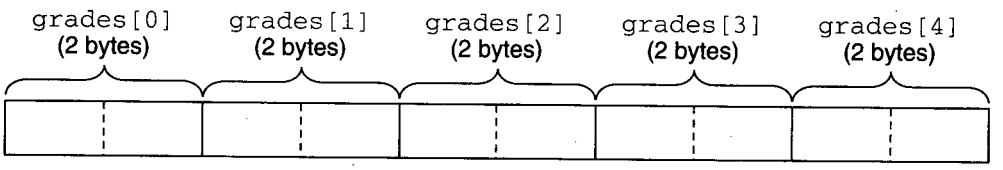
Using subscripts, the fourth element in the grades array is referred to as `grades[3]`. The use of a subscript, however, conceals the extensive use of addresses by the computer. Internally, the computer immediately uses the subscript to calculate the address of the desired element based on both the starting address of the array and the amount of storage used by each element. Calling the fourth element `grades[3]` forces the computer, internally, into the address computation

```
&grades[3] = &grades[0] + (3 * 2)
```

Remembering that the address operator, `&`, means "the address of," this last statement is read "the address of `grades[3]` equals the address of `grades[0]` plus 6." Figure 10–16 illustrates the address computation used to locate `grades[3]`.

Recall that a pointer is a variable used to store addresses. If we create a pointer to store the address of the first element in the `grades` array, we can mimic the

FIGURE 10–15 The `grades` Array in Storage



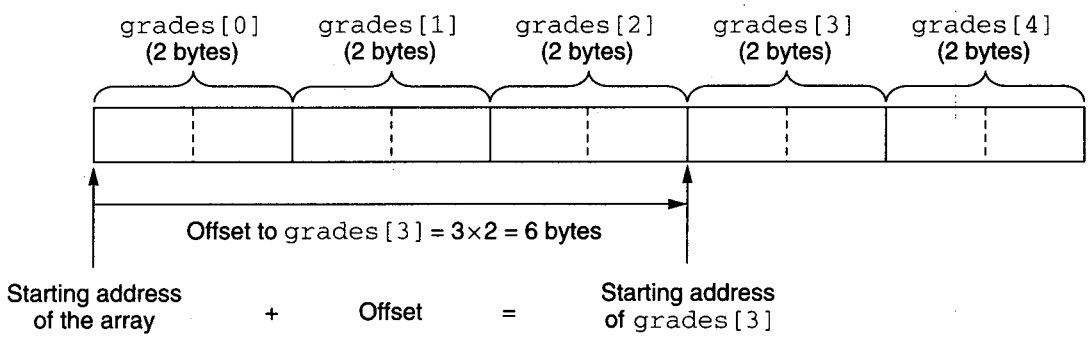


FIGURE 10–16 Using a Subscript to Obtain an Address

operation used by the computer to access the array elements. Before we do this, let us first consider Program 10-7.[3]

Program 10-7

```
#include <stdio.h>
main( )
{
  int i, grades[ ] = {98, 87, 92, 79, 85};

  for (i = 0; i <= 4; ++i)
    printf("\nElement %d is %d", i, grades[i]);
}
```

When Program 10-7 is run, the following display is obtained:

```
Element 0 is 98
Element 1 is 87
Element 2 is 92
Element 3 is 79
Element 4 is 85
```

Program 10-7 displays the values of the array `grades` using standard subscript notation. Now, let us store the address of array element `0` in a pointer. Then, using the indirection operator, `*`, we can use the address in the pointer to access each array element. For example, if we store the address of `grades[0]` into a pointer named `g_ptr` (using the assignment statement `g_ptr = &grades[0];`), then, as illustrated in Figure 10–17, the expression `*g_ptr`, which means "the variable pointed to by `g_pt`," references `grades[0]`.

[3] In traditional (non-ANSI) C, initializing the `grades` array requires declaring the array as `static`. Thus, in non-ANSI C the declaration for `grades` would have to be written as: `static int grades[ ] = {98, 87, 92, 79, 85};`.

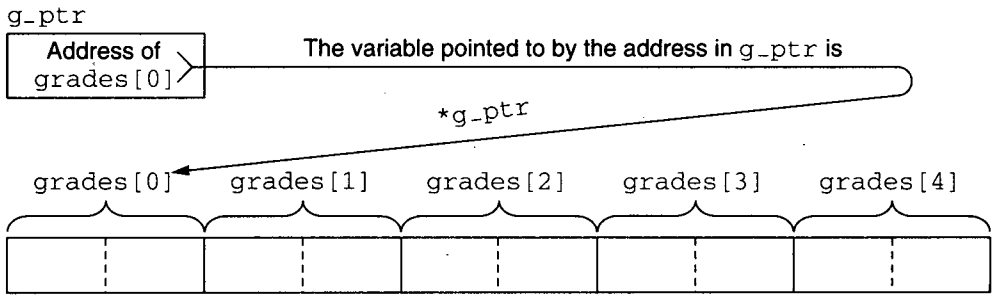


FIGURE 10–17 The Variable Pointed to by `*g_ptr` Is `grades[0]`

One unique feature of pointers is that *offsets* may be included in expressions using pointers. For example, the `1` in the expression `*(g_ptr + 1)` is an offset. The complete expression references the integer that is one beyond the variable pointed to by `g_ptr`. Similarly, as illustrated in Figure 10–18, the expression `*(g_ptr + 3)` references the variable that is three integers beyond the variable pointed to by `g_ptr`. This is the variable `grades[3]`.

Table 10–1 lists the complete correspondence between elements referenced by subscripts and by pointers and offsets, assuming that `g_ptr` has been initialized to `&grades[0]`. The relationships listed in Table 10–1 are illustrated in Figure 10–19.

Using the correspondence between pointers and subscripts illustrated in Figure 10–19, the array elements previously accessed in Program 10-7 using subscripts can now be accessed using pointers. This is done in Program 10-8.

Program 10-8

```
#include <stdio.h>
main( )
{
  int *g_ptr;          /* declare a pointer to an int */
  int i, grades[] = {98, 87, 92, 79, 85};

  g_ptr = &grades[0]; /* store the starting array address */
  for (i = 0; i <= 4; ++i)
    printf("\nElement %d is %d", i, *(g_ptr + i) );
}
```

The following display is obtained when Program 10-8 is run:

```
Element 0 is 98
Element 1 is 87
Element 2 is 92
Element 3 is 79
Element 4 is 85
```

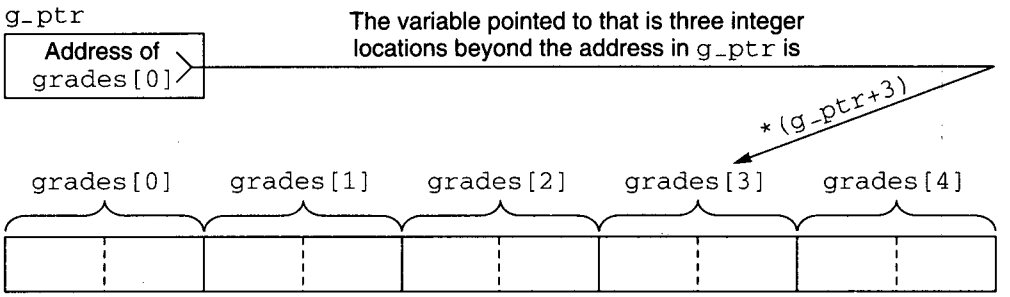


FIGURE 10–18 An Offset of 3 from the Address in `g_ptr`

TABLE 10–1 Array Elements May Be Referenced in Two Ways

Array Element	Subscript Notation	Pointer Notation
Element 0	`grades[0]`	`*g_ptr`
Element 1	`grades[1]`	`*(g_ptr + 1)`
Element 2	`grades[2]`	`*(g_ptr + 2)`
Element 3	`grades[3]`	`*(g_ptr + 3)`
Element 4	`grades[4]`	`*(g_ptr + 4)`

Notice that this is the same display produced by Program 10-7.

The method used in Program 10-8 to access individual array elements simulates how the computer internally references all array elements. Any subscript used by a programmer is automatically converted to an equivalent pointer expression by the computer. In our case, since the declaration of `g_ptr` included the information that integers are pointed to, any offset added to the address in

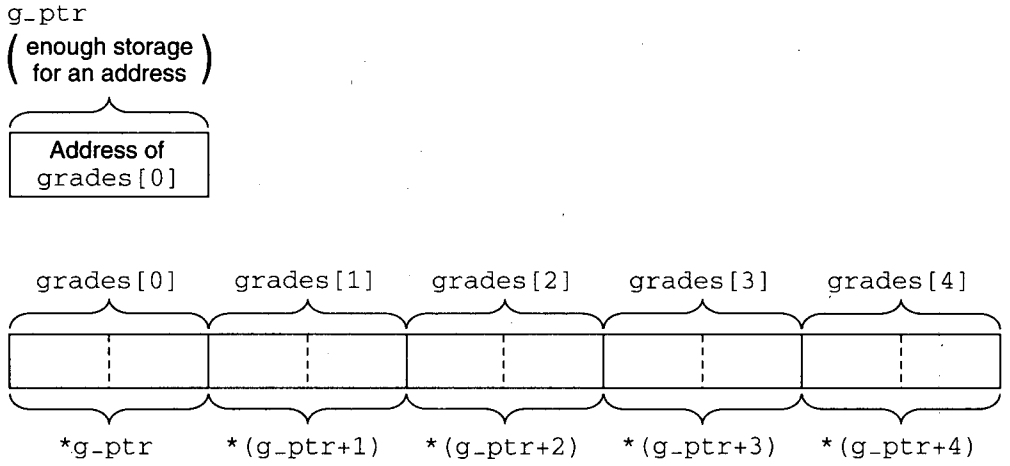


FIGURE 10–19 The Relationship Between Array Elements and Pointers

`g_ptr` is automatically scaled by the size of an integer. Thus, `*(g_ptr + 3)`, for example, refers to the address of `grades[0]` plus an offset of six bytes (3 * 2). This is the address of `grades[3]` illustrated in Figure 10–16.

The parentheses in the expression `*(g_ptr + 3)` are necessary to correctly reference the desired array element. Omitting the parentheses results in the expression `*g_ptr + 3`. This expression adds 3 to "the variable pointed to by `g_ptr`." Since `g_ptr` points to `grades[0]`, this expression adds the value of `grades[0]` and 3 together. Note also that the expression `*(g_ptr + 3)` does not change the address stored in `g_ptr`. Once the computer uses the offset to locate the correct variable from the starting address in `g_ptr`, the offset is discarded and the address in `g_ptr` remains unchanged.

Although the pointer `g_ptr` used in Program 10-8 was specifically created to store the starting address of the `grades` array, this was, in fact, unnecessary. When an array is created, the compiler automatically creates an internal pointer constant for it and stores the starting address of the array in this pointer. In almost all respects, a pointer constant is identical to a pointer variable created by a programmer; but, as we shall see, there are some differences.

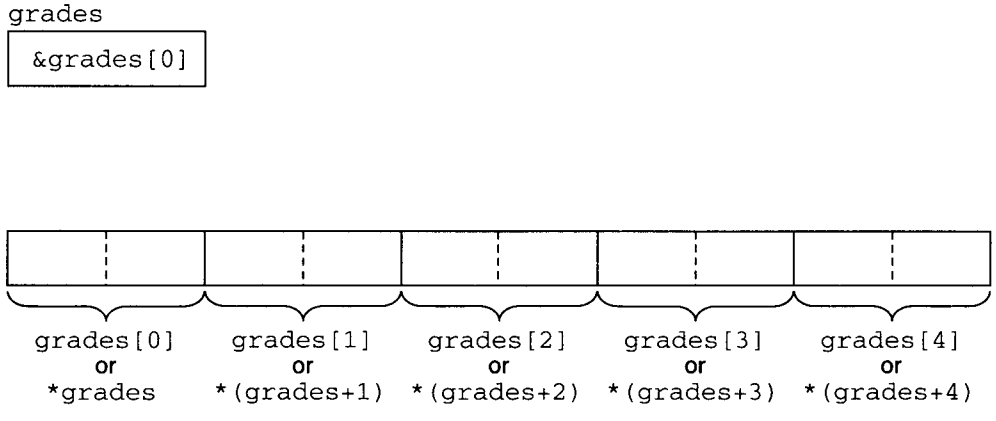
For each array created, the name of the array becomes the name of the pointer constant created by the compiler for the array, and the starting address of the first location reserved for the array is stored in this pointer. Thus, declaring the `grades` array in both Program 10-7 and Program 10-8 actually reserved enough storage for five integers, created an internal pointer named `grades`, and stored the address of `grades[0]` in the pointer. This is illustrated in Figure 10–20.

The implication is that every reference to `grades` using a subscript can be replaced by an equivalent reference using `grades` as a pointer. Thus, wherever the expression `grades[i]` is used, the expression `*(grades + i)` can also be used. This is illustrated in Program 10-9, where `grades` is used as a pointer to reference all of its elements.

Executing Program 10-9 produces the same output previously produced by Program 10-7 and Program 10-8. However, using `grades` as a pointer made it unnecessary to declare and initialize the pointer `g_ptr` used in Program 10-8.

In most respects an array name and pointer can be used interchangeably. *A true pointer, however, is a variable and the address stored in it can be changed. An array name is a pointer constant and the address stored in the pointer cannot be changed by an*

FIGURE 10–20 Creating an Array Also Creates a Pointer



Program 10-9

```
#include <stdio.h>
main( )
{
  int i, grades[ ] = {98, 87, 92, 79, 85};

  for (i = 0; i <= 4; ++i)
    printf("\nElement %d is %d", i, *(grades + i) );
}
```

assignment statement. Thus, a statement such as `grades = &grades[2];` is invalid. This should come as no surprise. Since the whole purpose of an array name is to correctly locate the beginning of the array, allowing a programmer to change the address stored in the array name would defeat this purpose and lead to havoc whenever array elements were referenced. Also, expressions taking the address of an array name are invalid because the pointer created by the compiler is internal to the computer, not stored in memory as are pointer variables. Thus, trying to take the address of `grades` using the expression `&grades` results in a compiler error.

An interesting sidelight to the observation that elements of an array can be referenced using pointers is that a pointer reference can always be replaced with a subscript reference. For example, if `num_ptr` is declared as a pointer variable, the expression `*(num_ptr + i)` can also be written as `num_ptr[i]`. This is true even though `num_ptr` is not created as an array. As before, when the compiler encounters the subscript notation, it replaces it internally with the pointer notation.

Exercises 10.3

1. Replace each of the following references to a subscripted variable with a pointer reference.

a. `prices[5]`	*b.* `grades[2]`	*c.* `yield[10]`
d. `dist[9]`	*e.* `mile[0]`	*f.* `temp[20]`
g. `celsius[16]`	*h.* `num[50]`	*i.* `time[12]`

2. Replace each of the following references using a pointer with a subscript reference.

a. `*(message + 6)`	*b.* `*amount`	*c.* `*(yrs + 10)`
d. `*(stocks + 2)`	*e.* `*(rates + 15)`	*f.* `*(codes + 19)`

3 a. List the three things that the declaration statement `double prices[5];` causes the compiler to do.

b. If each double precision number uses four bytes of storage, how much storage is set aside for the `prices` array?

c. Draw a diagram similar to Figure 10–20 for the `prices` array.

d. Determine the byte offset relative to the start of the `prices` array, corresponding to the offset in the expression `*(prices + 3)`.

4 *a.* Write a declaration to store the string "This is a sample" into an array named samtest. Include the declaration in a program that displays the values in samtest using a for loop and pointer references to each element in the array.
b. Modify the program written in Exercise 4a to display only array elements 10 through 15 (these are the letters s, a, m, p, l, and e).

5. Write a declaration to store the following values into an array named rates: 12.9, 18.6, 11.4, 13.7, 9.5, 15.2, 17.6. Include the declaration in a program that displays the values in the array using pointer notation.

6. Repeat Exercise 6 in Section 8.1, but use pointer references to access all array elements.

7. Repeat Exercise 7 in Section 8.1, but use pointer references to access all array elements.

10.4 Pointer Arithmetic

Pointer variables, like all variables, contain values. The value stored in a pointer is, of course, an address. Thus, by adding and subtracting numbers to pointers we can obtain different addresses. Additionally, the addresses in pointers can be compared using any of the relational operators (==, !=, <, >, etc.) that are valid for comparing other variables. In performing arithmetic on pointers we must be careful to produce addresses that point to something meaningful. In comparing pointers we must also make comparisons that make sense. Consider these declarations:

```
int nums[100];

int *n_pt;
```

To set the address of nums[0] into n_pt, either of the following two assignment statements can be used:

```
n_pt = &nums[0];

n_pt = nums;
```

The two assignment statements produce the same result because nums is a pointer constant that itself contains the address of the first location in the array. This is, of course, the address of nums[0]. Figure 10–21 illustrates the allocation of memory resulting from the previous declaration and assignment statements, assuming that each integer requires two bytes of memory and that the location of the beginning of the nums array is at address 18943.

Once n_pt contains a valid address, values can be added and subtracted from the address to produce new addresses. When adding or subtracting numbers to pointers, the computer automatically adjusts the number to ensure that the result still "points to" a value of the correct type. For example, the statement n_pt = n_pt + 4; forces the computer to scale the 4 by the correct number to ensure that the resulting address is the address of an integer. Assuming that each integer requires two bytes of storage, as illustrated in Figure 10–21, the computer multiplies the 4 by two and then adds eight to the address in n_pt. The resulting address is 18942, which is the correct address of nums[4].

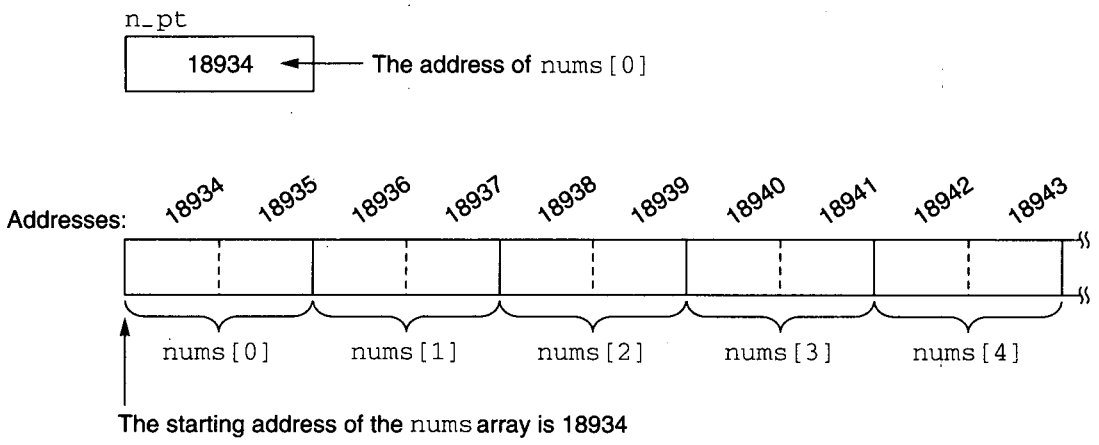
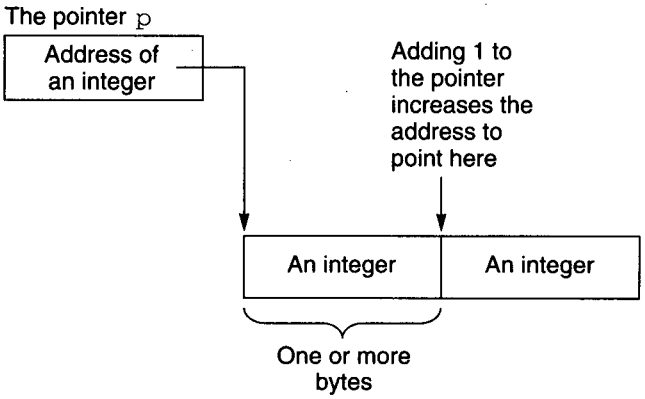


FIGURE 10–21 The nums Array in Memory

This automatic scaling by the computer ensures that the expression `n_pt + i`, where `i` is any positive integer, correctly points to the ith element beyond the one currently being pointed to by `n_pt`. Thus, if `n_pt` initially contains the address of `nums[0]`, `n_pt + 4` is the address of `nums[4]`, `n_pt + 50` is the address of `nums[50]`, and `n_pt + i` is the address of `nums[i]`. Although we have used actual addresses in Figure 10–21 to illustrate the scaling process, the programmer need never know or care about the actual addresses used by the computer. The manipulation of addresses using pointers generally does not require knowledge of the actual address.

Addresses can also be incremented or decremented using both prefix and postfix increment and decrement operators. Adding one to a pointer causes the pointer to point to the next element of the type being pointed to. Decrementing a pointer causes the pointer to point to the previous element. For example, if the pointer variable `p` is a pointer to an integer, the expression `p++` causes the address in the pointer to be incremented to point to the next integer. This is illustrated in Figure 10–22.

FIGURE 10–22 Increments Are Scaled When Used with Pointers



In reviewing Figure 10–22, notice that the increment added to the pointer is correctly scaled to account for the fact that the pointer is used to point to integers. It is, of course, up to the programmer to ensure that the correct type of data is stored in the new address contained in the pointer.

The increment and decrement operators can be applied as both prefix and postfix pointer operators. All of the following combinations using pointers are valid:

```
*pt_num++      /* use the pointer and then increment it */
*++pt_num      /* increment the pointer before using it */
*pt_num--      /* use the pointer and then decrement it */
*--pt_num      /* decrement the pointer before using it */
```

Of the four possible forms, the most commonly used is the form `*pt_num++`. This is because such an expression allows each element in an array to be accessed as the address is "marched along" from the starting address of the array to the address of the last array element. To see the use of the increment operator, consider Program 10-10. In this program each element in the `nums` array is retrieved by successively incrementing the address in `n_pt`.[4]

Program 10-10

```
#include <stdio.h>
main( )
{
  int nums[5] = {16, 54, 7, 43, -5};
  int i, total = 0, *n_pt;

  n_pt = nums;     /* store address of nums[0] in n_pt */
  for (i = 0; i <= 4; ++i)
    total = total + *n_pt++;

  printf("The total of the array elements is %d", total);
}
```

The output produced by Program 10-10 is:

```
The total of the array elements is 115
```

The expression `total = total + *n_pt++` used in Program 10-10 is a standard accumulating expression. Within this expression, the term `*n_pt++`

[4] If you are using a non-ANSI C compiler, the declaration statement for the `nums` array must be changed to: `static int nums[5] = {16, 54, 7, 43, -5};`. This is because `static` local arrays may be initialized at compile time, whereas `automatic` local arrays cannot.

first causes the computer to retrieve the integer pointed to by n_pt. This is done by the *n_pt part of the term. The postfix increment, ++, then adds one to the address in n_pt so that n_pt now contains the address of the next array element. The increment is, of course, scaled by the computer so that the actual address in n_pt is the correct address of the next element.

Pointers may also be compared. This is particularly useful when dealing with pointers that point to elements in the same array. For example, rather than using a counter in a for loop to correctly access each element in an array, the address in a pointer can be compared to the starting and ending address of the array itself. The expression

```
n_pt <= &nums[4]
```

is true (nonzero) as long as the address in n_pt is less than or equal to the address of nums[4]. Since nums is a pointer constant that contains the address of nums[0], the term &nums[4] can be replaced by the equivalent term nums + 4. Using either of these forms, Program 10-10 can be rewritten in Program 10-11 to continue adding array elements while the address in n_pt is less than or equal to the address of the last array element.

Program 10-11

```
#include <stdio.h>
main( )
{
  int nums[5] = {16, 54, 7, 43, -5};
  int total = 0, *n_pt;

  n_pt = nums;     /* store address of nums[0] in n_pt */

  while (n_pt <= nums + 4)
    total += *n_pt++;

  printf("The total of the array elements is %d", total);
}
```

Notice that in Program 10-11 the compact form of the accumulating expression, total += *n_pt++, was used in place of the longer form, total = total + *n_pt++. Also, the expression nums + 4 does not change the address in nums. Since nums is an array name and not a pointer variable, its value cannot be changed. The expression nums + 4 first retrieves the address in nums, adds 4 to this address (appropriately scaled), and uses the result for comparison purposes. Expressions such as *nums++, which attempt to change the address, are invalid. Expressions such as *nums or *(nums + i), which use the address without attempting to alter it, are valid.

Pointer Initialization

Like all variables, pointers can be initialized when they are declared. When initializing pointers, however, you must be careful to set an address in the pointer. For example, an initialization such as

```
int *pt_num = &miles;
```

is only valid if `miles` itself were declared as an integer variable prior to `pt_num`. Here we are creating a pointer to an integer and setting the address in the pointer to the address of an integer variable. Notice that if the variable `miles` is declared subsequently to `pt_num`, as follows,

```
int *pt_num = &miles;
int miles;
```

an error occurs. This is because the address of `miles` is used before `miles` has even been defined. Since the storage area reserved for `miles` has not been allocated when `pt_num` is declared, the address of `miles` does not yet exist.

Pointers to arrays can also be initialized within their declaration statements. For example, if `prices` has been declared an array of floating point numbers, either of the following two declarations can be used to initialize the pointer named `zing` to the address of the first element in `prices`:

```
float *zing = &prices[0];

float *zing = prices;
```

The last initialization is correct because `prices` is itself a pointer constant containing an address of the proper type. (The variable name `zing` was selected in this example to reinforce the idea that any variable name can be selected for a pointer.)

Exercises 10.4

1. Replace the `while` statement in Program 10-11 with a `for` statement.

2 *a.* Write a C program that stores the following numbers in the array named `rates`: 6.25, 6.50, 6.8, 7.2, 7.35, 7.5, 7.65, 7.8, 8.2, 8.4, 8.6, 8.8, 9.0. Display the values in the array by changing the address in a pointer called `disp_pt`. Use a `for` statement in your program.

b. Modify the program written in Exercise 2a to use a `while` statement.

3 *a.* Write a program that stores the string `Hooray for All of Us` into an array named `strng`. Use the declaration `strng[] = "Hooray for All of Us";`, which ensures that the end-of-string escape sequence `\0` is included in the array. Display the characters in the array by changing the address in a pointer called `mess_pt`. Use a `for` statement in your program.

b. Modify the program written in Exercise 3a to use the `while` statement:

```
while (*mess_pt++ != '\0').
```

c. Modify the program written in Exercise 3a to start the display with the word `All`.

4. Write a C program that stores the following numbers in the array named `miles`: 15, 22, 16, 18, 27, 23, 20. Using pointers have your program copy the data stored in `miles` to another array named `dist` and then display the values in the `dist` array.

5. Write a program that stores the following letters in the array named `message`: `This is a test`. Using pointers have your program copy the data stored in `message` to another array named `mess2` and then display the letters in the `mess2` array.

6. Write a program that declares three single-dimensional arrays named `miles`, `gallons`, and `mpg`. Each array should be capable of holding ten elements. In the `miles` array store the numbers 240.5, 300.0, 189.6, 310.6, 280.7, 216.9, 199.4, 160.3, 177.4, 192.3. In the `gallons` array store the numbers 10.3, 15.6, 8.7, 14, 16.3, 15.7, 14.9, 10.7, 8.3, 8.4. Each element of the `mpg` array should be calculated as the corresponding element of the `miles` array divided by the equivalent element of the `gallons` array; for example, `mpg[0] = miles[0] / gallons[0]`. Use pointers to calculate and display the elements of the `mpg` array.

10.5 Passing and Using Array Addresses

When an array is passed to a function, its address is the only item actually passed. By this we mean the address of the first location used to store the array, as illustrated in Figure 10–23. Since the first location reserved for an array corresponds to element 0 of the array, the "address of the array" is also the address of element 0.

For a specific example in which an array is passed to a function, let us consider Program 10-12. In this program, the `nums` array is passed to the `find_max( )` function using conventional array notation.

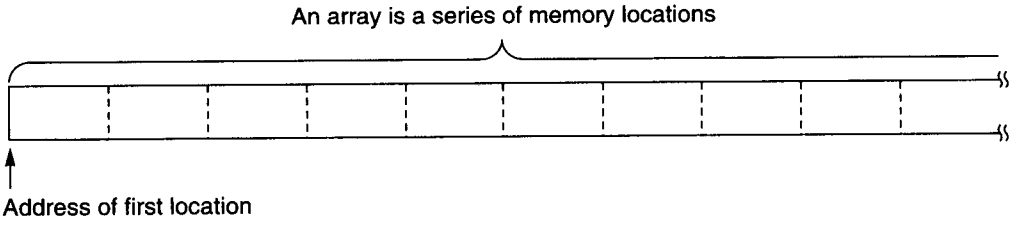
The output displayed when Program 10-12 is executed is:

```
The maximum value is 27
```

The argument named `vals` in the header line declaration for `find_max( )` actually receives the address of the array `nums`. As such, `vals` is really a pointer, since pointers are variables (or arguments) used to store addresses. Since the address passed into `find_max( )` is the address of an integer, another suitable header line for `find_max( )` is:

```
find_max(int *vals, int num_els)
/* here vals is declared as a pointer to an integer */
```

FIGURE 10–23 The Address of an Array Is the Address of the First Location Reserved for the Array



Program 10-12

```
#include <stdio.h>
main( )
{
  int nums[5] = {2, 18, 1, 27, 16};
  int find_max(int [ ], int);     /* function prototype */

  printf("Themaximumvalueis %d", find_max(nums,5));
}

int find_max(int vals[ ], int num_els)    /* find the maximum value */
{
  int i, max = vals[0];

  for (i = 1; i < num_els; ++i)
   if (max < vals[i]) max = vals[i];

  return(max);
}
```

The declaration `int *vals` in the header line declares that `vals` is used to store an address of an integer. The address stored is, of course, the location of the beginning of an array. The following is a rewritten version of the `find_max( )` function that uses the new pointer declaration for `vals`, but retains the use of subscripts to refer to individual array elements:

```
int find_max(int *vals, int num_els)    /* find the maximum value */
{
  int i, max = vals[0];

  for (i = 1; i < num_els; ++i)
   if (max < vals[i]) max = vals[i];

  return(max);
}
```

Regardless of how `vals` is declared in the function header or how it is used within the function body, it is truly a pointer variable. Thus, the address in `vals` may be modified. This is not true for the name `nums`. Since `nums` is the name of the originally created array, it is a pointer constant. As described in Section 10-1, this means that the address in `nums` cannot be changed and that the address of `nums` itself cannot be taken. No such restrictions, however, apply to the pointer variable named `vals`. All the address arithmetic that we learned in the previous section can be legitimately applied to `vals`.

We shall write two additional versions of `find_max( )`, both using pointers instead of subscripts. In the first version we simply substitute pointer notation

for subscript notation. In the second version we use address arithmetic to change the address in the pointer.

As previously stated, a reference to an array element using the subscript notation `array_name[i]` can always be replaced by the pointer notation `*(array_name + i)`. In our first modification to `find_max( )`, we make use of this correspondence by simply replacing all references to `vals[i]` with the equivalent expression `*(vals + i)`.

```
int find_max(int *vals, int num_els)    /* find the maximum value */
{
  int i, max = *vals;

  for (i = 1; i < num_els; ++i)
   if (max < *(vals + i) )  max = *(vals + i);

  return(max);
}
```

Our second version of `find_max( )` makes use of the fact that the address stored in `vals` can be changed. After each array element is retrieved using the address in `vals`, the address itself is incremented by one in the altering list of the `for` statement. The expression `*vals++` used initially to set `max` to the value of `vals[0]` also adjusts the address in `vals` to point to the second element in the array. The element obtained from this expression is the array element pointed to by `vals` before `vals` is incremented. The postfix increment, `++`, does not change the address in `vals` until after the address has been used to retrieve the array element.

```
int find_max(int *vals, int num_els)    /* find the maximum value */
{
  int i, max = *vals++;    /* get the first element and increment */

  for (i = 1; i < num_els; ++i, ++vals)
  {
    if (max < *vals)  max = *vals;
  }
  return(max);
}
```

Let us review this version of `find_max( )`. Initially the maximum value is set to "the thing pointed to by `vals`." Since `vals` initially contains the address of the first element in the array passed to `find_max( )`, the value of this first element is stored in `max`. The address in `vals` is then incremented by one. The one that is added to `vals` is automatically scaled by the number of bytes used to store integers. Thus, after the increment, the address stored in `vals` is the address of the next array element. This is illustrated in Figure 10–24. The value of this next element is compared to the maximum and the address is again incremented, this time from within the altering list of the `for` statement. This process continues until all the array elements have been examined.

The version of `find_max( )` that appeals to you is a matter of personal style and taste. Generally, beginning programmers feel more at ease using subscripts

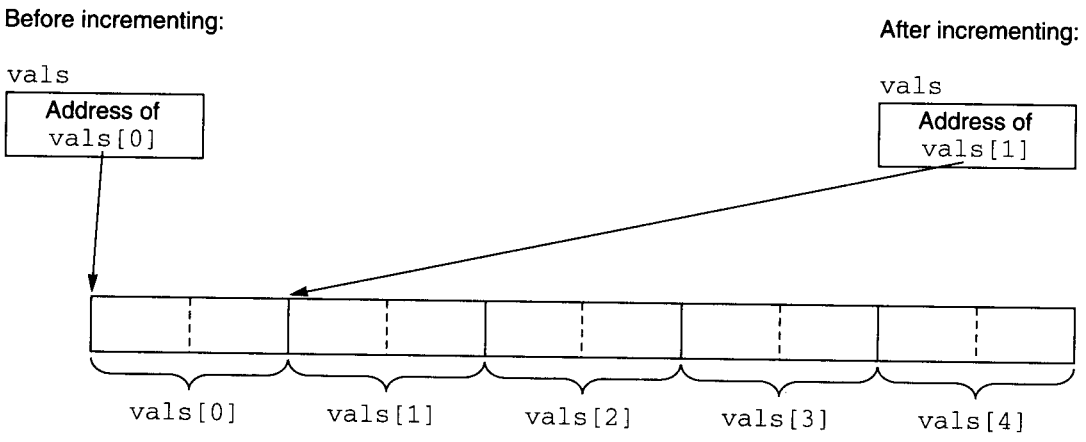


FIGURE 10–24 Pointing to Different Elements

than using pointers. Also, if the program uses an array as the natural storage structure for the application and data at hand, an array reference using subscripts is more appropriate to clearly indicate the intent of the program. However, as we learn about strings and data structures, we realize that the use of pointers is an increasingly useful and powerful tool in its own right. In these instances there is no simple or easy equivalence using subscripts.

One further "neat trick" can be gleaned from our discussion. Since passing an array to a function really involves passing an address, we can just as well pass any valid address. For example, the function call `find_max(&nums[2],3)` passes the address of `nums[2]` to `find_max( )`. Within `find_max( )` the pointer `vals` stores the address and the function starts the search for a maximum at the element corresponding to this address. Thus, from `find_max( )`'s perspective, it has received an address and proceeds appropriately.

Advanced Pointer Notation[5]

Access to multidimensional arrays can also be made using pointer notation, although the notation becomes more and more cryptic as the array dimensions increase. An extremely useful application of this notation occurs with two-dimensional character arrays, one of the topics of the next chapter. Here we consider pointer notation for two-dimensional numeric arrays. For example, consider the declaration

```
int nums[2][3] = { {16,18,20},
                   {25,26,27} };
```

This declaration creates an array of elements and a set of pointer constants named `nums`, `nums[0]`, and `nums[1]`. The relationship between these pointer constants and the elements of the `nums` array are illustrated in Figure 10–25.

The availability of the pointer constants associated with a two-dimensional array allows us to reference array elements in a variety of ways. One way is to

[5] This topic may be omitted with no loss of subject continuity.

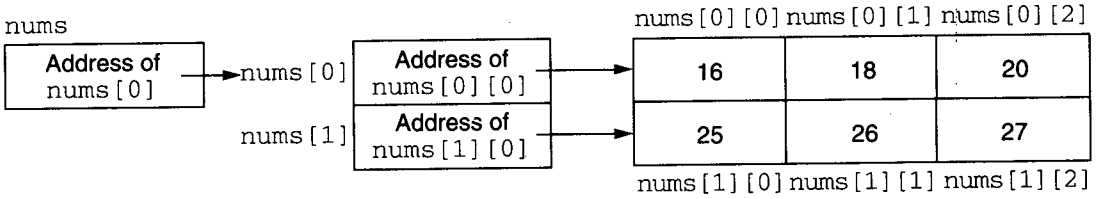


FIGURE 10–25 Storage of the nums Array and Associated Pointer Constants

consider the two-dimensional array as an array of rows, where each row is itself an array of three elements. Considered in this light, the address of the first element in the first row is provided by `nums[0]` and the address of the first element in the second row is provided by `nums[1]`. Thus, the variable pointed to by `nums[0]` is `nums[0][0]` and the variable pointed to by `nums[1]` is `nums[1][0]`. Once the nature of these constants is understood, each element in the array can be accessed by applying an appropriate offset to the appropriate pointer. Thus, the following notations are equivalent:

Pointer Notation	Subscript Notation	Value
`*nums[0]`	`nums[0][0]`	16
`*(nums[0] + 1)`	`nums[0][1]`	18
`*(nums[0] + 2)`	`nums[0][2]`	20
`*nums[1]`	`nums[1][0]`	25
`*(nums[1] + 1)`	`nums[1][1]`	26
`*(nums[1] + 2)`	`nums[1][2]`	27

We can now go even further and replace `nums[0]` and `nums[1]` with their respective pointer notations, using the address of `nums` itself. As illustrated in Figure 10–25, the variable pointed to by `nums` is `nums[0]`. That is, `*nums` is `nums[0]`. Similarly, `*(nums + 1)` is `nums[1]`. Using these relationships leads to the following equivalences:

Pointer Notation	Subscript Notation	Value
`*(*nums)`	`nums[0][0]`	16
`*(*nums + 1)`	`nums[0][1]`	18
`*(*nums + 2)`	`nums[0][2]`	20
`*(*(nums + 1))`	`nums[1][0]`	25
`*(*(nums + 1) + 1)`	`nums[1][1]`	26
`*(*(nums + 1) + 2)`	`nums[1][2]`	27

The same notation applies when a two-dimensional array is passed to a function. For example, assume that the two-dimensional array `nums` is passed to the function `calc( )` using the call `calc(nums);`. Here, as with all array passes, an address is passed. A suitable function header line for the function `calc( )` is:

```
calc(int pt[2][3])
```

As we have already seen, the argument declaration for `pt` can also be:

```
calc(int pt[ ][3])
```

Using pointer notation, another suitable declaration is:

```
calc(int (*pt)[3])
```

In this last declaration the inner parentheses are required to create a single pointer to objects of three integers. Each object is, of course, equivalent to a single row of the `nums` array. By suitably offsetting the pointer, each element in the array can be accessed. Notice that without the parentheses the declaration becomes

```
int *pt[3]
```

which creates an array of three pointers, each one pointing to a single integer.

Once the correct declaration for `pt` is made (any of the three valid declarations can be used), the following notations within the function `calc( )` are all equivalent:

Pointer Notation	Subscript Notation	Value
`*(*pt)`	`pt[0][0]`	16
`*(*pt+1)`	`pt[0][1]`	18
`*(*pt+2)`	`pt[0][2]`	20
`*(*(pt+1))`	`pt[1][0]`	25
`*(*(pt+1)+1)`	`pt[1][1]`	26
`*(*(pt+1)+2)`	`pt[1][2]`	27

The last two notations using pointers are encountered in more advanced C programs. The first of these occurs because functions can return any valid C scalar data type, including pointers to any of these data types. If a function returns a pointer, the data type being pointed to must be declared in the function's declaration. For example, the declaration

```
int *calc( )
```

declares that `calc( )` returns a pointer to an integer value. This means that an address of an integer variable is returned. Similarly, the declaration

```
float *taxes( )
```

declares that `taxes( )` returns a pointer to a floating point value. This means that an address of a floating point variable is returned.

In addition to declaring pointers to integers, floating point numbers, and C's other data types, pointers can also be declared that point to (contain the address of) a function. Pointers to functions are possible because function names, like array names, are themselves pointer constants. For example, the declaration

```
int (*calc)( )
```

declares `calc( )` to be a pointer to a function that returns an integer. This means that `calc` will contain the address of a function, and the function whose address is in the variable `calc` returns an integer value. If, for example, the function `sum( )` returns an integer, the assignment `calc = sum;` is valid.

Exercises 10.5

1. The following declaration was used to create the `prices` array:

```
double prices[500];
```

Write three different function header lines for a function named `sort_arr( )` that accepts the `prices` array as an argument named `in_array` and returns an integer.

2. The following declaration was used to create the `keys` array:

```
char keys[256];
```

Write three different function header lines for a function named `find_key( )` that accepts the `keys` array as an argument named `select` and returns an integer.

3. The following declarations was used to create the `rates` array:

```
float rates[256];
```

Write three different function header lines for a function named `prime( )` that accepts the `rates` array as an argument named `rates` and returns an integer.

4. Modify the `find_max( )` function to locate the minimum value of the passed array. Write the function using only pointers.

5. In the last version of `find_max( )` presented, `vals` was incremented inside the altering list of the `for` statement. Instead, suppose that the incrementing was done within the condition expression of the `if` statement, as follows:

```
find_max(vals,num_els)     /* incorrect version          */
int *vals;                 /* vals declared as a pointer */
in num_els;
{
  int i, max = *vals++;    /* get the first element and increment */

  for (i = 1; i < (num_els; ++i)
  {
    if (max < *vals++)  max = *vals;
  }
  return(max);
}
```

This version produces an incorrect result. Determine why.

6 a. Write a program that has a declaration in `main( )` to store the following numbers into an array named `rates`: 6.5, 7.2, 7.5, 8.3, 8.6, 9.4, 9.6, 9.8, 10.0. There should be a function call to `show( )` that accepts `rates` in an argument named `rates` and then displays the numbers using the pointer notation `*(rates + i)`.

b. Modify the `show( )` function written in Exercise 6a to alter the address in `rates`.

Use the expression `*rates++` rather than `*(rates + i)` to retrieve the correct element.

7 *a.* Write a program that has a declaration in `main( )` to store the string `Vacation is near` into an array named `message`. There should be a function call to `display( )` that accepts `message` in an argument named `strng` and then displays the message using the pointer notation `*(strng + i)`.

b. Modify the `display( )` function written in Exercise 7a to alter the address in `message`. Use the expression `*strng++ rather than *(strng + i) to retrieve the correct element.`

8. Write a program that declares three single-dimensional arrays named `price`, `quantity`, and `amount`. Each array should be declared in `main( )` and be capable of holding ten double precision numbers. The numbers to be stored in price are 10.62, 14.89, 13.21, 16.55, 18.62, 9.47, 6.58, 18.32, 12.15, 3.98. The numbers to be stored in quantity are 4, 8.5, 6, 7.35, 9, 15.3, 3, 5.4, 2.9, 4.8. Have your program pass these three arrays to a function called `extend( )`, which calculates the elements in the `amount` array as the product of the equivalent elements in the `price` and `quantity` arrays (for example, `amount[1] = price[1] * quantity[1]`).

After `extend( )` has put values into the `amount` array, display the values in the array from within `main( )`. Write the `extend( )` function using pointers.

9 *a.* Determine the output of the following program:

```
main( )
{
  int nums[2][3] = { {33,16,29},
                     {54,67,99}};
  void arr(int[][]);
  arr(nums);
}

void arr(int (*val)[3])
{
    printf("\n %d",*(*val) );
    printf("\n %d",*(*val + 1) );
    printf("\n %d",*(*(val + 1) + 2) );
    printf("\n %d",*(*val) + 1 );
}
```

b. Given the declaration for `val` in the `arr( )` function, would the reference `val[1][2]` be valid within the function?

10.6 Passing and Returning File Names

File names are passed to a function using the same procedures for passing all function arguments. For passing a file name this requires declaring the passed argument as a pointer to a `FILE`. For example, in Program 10-13 a file named `out_file` is opened in `main( )` and the file name is passed to the function `in_out( )`, which is then used to write five lines of user-entered text to the file.

Within `main( )` the file is known as `out_file`. The value in `out_file`, which is an address, is passed to the `in_out( )` function. The function `in_out( )`

Program 10-13

```
#include <stdio.h>
main( )
{
  FILE *out_file;
  void in_out(FILE *);  /* function prototype  */

  out_file = fopen("prices.dat","w");
  in_out(out_file);
  fclose(out_file);
}

void in_out(FILE *fname)    /* fname is a pointer to a FILE */
{
  int count;
  char line[81];  /* enough storage for one line of text */

  printf("Please enter five lines of text:\n");
  for (count = 0; count < 5; ++count)
  {
    gets(line);
    fprintf(fname,"%s\n",line);
  }
  return;
}
```

stores the address in the argument named `fname` and correctly declares `fname` to be a pointer to a `FILE`. Notice that the function prototype for `in_out( )` within `main( )` declares that `in_out( )` expects to receive a pointer to a `FILE`.

Returning a file name from a function also requires following the same rules used to return any value from a function. This means including the data type of the returned value in the function header, making sure the correct variable type is actually returned from the function, and alerting the calling function to the returned data type. For example, assume that the function `get_open( )` is called with no passed arguments. The purpose of the function is to prompt a user for a file name, open the file for output, and pass the file name back to the calling function. Since `get_open( )` returns a file name that is actually a pointer to a `FILE`, the correct function prototype for `get_open( )` is:

```
FILE *get_open(void);
```

This prototype specifically declares that the function `get_open( )` expects no arguments and will return a pointer to a `FILE`. It is consistent with the pointer declarations that have been made previously.

Once a function has been declared to return a pointer to a `FILE`, there must be at least one variable or argument in the function consistent with this declara-

tion that can be used for the actual returned value. Consider Program 10-14. In this program, `get_open( )` returns a file name to `main( )`.

Program 10-14

```
#include <stdio.h>
main( )
{
  FILE  *get_open(void), *out_file;
  void in_out(FILE *);   /* a filename will be passed into in_out */

  out_file = get_open();
  in_out(out_file);
  fclose(out_file);
}

FILE *get_open(void)    /* get_open( ) returns a pointer to a FILE */
{
  FILE  *fname;
  char name[13];

  printf("\nEnter a file name: ");
  gets(name);
  fname = fopen(name,"w");
  return(fname);
}

void in_out(FILE *fname)  /* fname is a pointer to a FILE */
{
  int count;
  char line[81];  /* enough storage for one line of text */

  printf("Please enter five lines of text:\n");
  for (count = 0; count < 5; ++count)
  {
    gets(line);
    fprintf(fname,"%s\n",line);
  }
}
```

Program 10-14 is simply a modified version of Program 10-13 that now allows the user to enter a file name from the standard input device. Although the function `get_open( )` is in "bare bones" form, it does illustrate the correct function declaration for returning a file name. The `get_open( )` function declaration defines the function as returning a pointer to a `FILE`. Within `get_open( )`, the returned variable, `fname`, is the correct data type. Finally, since `get_open( )` is

defined in the program after `main( )`, `main( )` is alerted to the returned value by the inclusion of a declaration statement for the `get_open` function.

`get_open( )` is a "bare bones" function in that it does no checking on the file being opened for output. If the name of an existing data file is entered, the file will be destroyed when it is opened in write mode. A useful "trick" to prevent this type of mishap is to open the entered file name in read mode. Then, if the file exists, the `fopen( )` function returns a nonzero pointer to indicate that the file is available for input. This can be used to alert the user that a file with the entered name currently exists in the system and to request confirmation that the data in the file can be destroyed and the file name used for the new output file. Before the file can be reopened in write mode, of course, it would have to be closed. The implementation of this algorithm is left as an exercise.

Exercises 10.6

1. A function named `p_file( )` is to receive a file name as an argument. What declarations are required to pass a file name to `p_file( )`?

2 *a.* A function name `get_file( )` is to return a file name. What declarations are required in the function header and internal to the file?
b. What declaration statement is required in each function that calls `get_file( )` to ensure correct receipt of the file name returned by `get_file( )`? Under what conditions can this declaration be omitted?

3. Write a function named `fcheck( )` that checks whether a file exists. The function should be passed a file name. If the file exists, the function should return a value of 1, otherwise the function should return a value of zero.

4. Rewrite the function `get_open( )` used in Program 10-14 to incorporate the file-checking procedures described in the text. Specifically, if the entered file name exists, an appropriate message should be displayed. The user should then be presented with the option of entering a new file name or allowing the program to overwrite the existing file, append to it, or exit.

10.7 Common Programming Errors

In using the material presented in this chapter, be aware of the following possible errors:

1. Attempting to store an address in a variable that has not been declared as a pointer.
2. Using a pointer to reference nonexistent array elements. For example, if `nums` is an array of ten integers, the expression `*(nums + 15)` points six integer locations beyond the last element of the array. As C does not do any bounds checking on array references, this type of error is not caught by the compiler. This is the same error, disguised in pointer notation form, as using a subscript to reference an out-of-bounds array element.

3. Incorrectly applying the address and indirection operators. For example, if `pt` is a pointer variable, the expressions:

```
pt = &45

pt = &(miles + 10)
```

are both invalid because they attempt to take the address of a value. Notice that the expression `pt = &miles + 10`, however, is valid. Here, `10` is added to the address of `miles`. Again, it is the programmer's responsibility to ensure that the final address "points to" a valid data element.

4. Taking addresses of register variables. Thus, for the declarations

```
register int total;
int *pt_tot;
```

the assignment

```
pt_tot = &total;
```

is invalid. The reason for this is that register variables are stored in a computer's internal registers, and these storage areas do not have standard memory addresses.

5. Taking addresses of pointer constants. For example, given the declarations

```
int nums[25];
int *pt;
```

the assignment

```
pt = &nums;
```

is invalid. `nums` is a pointer constant that is itself equivalent to an address. The correct assignment is `pt = nums`.

6. Initializing pointer variables incorrectly. For example, the initialization

```
int *pt = 5;
```

is invalid. Since `pt` is a pointer to an integer, it must be initialized with a valid address.

7. Becoming confused about whether a variable *contains* an address or *is* an address. Pointer variables and pointer arguments contain addresses. Although a pointer constant is synonymous with an address, it is useful to treat pointer constants as pointer variables with two restrictions:

- The address of a pointer constant cannot be taken.
- The address "contained in" the pointer constant cannot be altered.

Except for these two restrictions, pointer constants and variables can be used almost interchangeably. Therefore, when an address is required any of the following can be used:

a pointer variable name
a pointer argument name
a pointer constant name
a nonpointer variable name preceded by the address operator (e.g., `&variable`)
a nonpointer argument name preceded by the address operator (e.g., `&argument`)

Some of the confusion surrounding pointers is caused by the cavalier use of the word *pointer*. For example, the phrase "a function requires a pointer argument" is more clearly understood when it is realized that the phrase really means "a function requires an address as an argument." Similarly, the phrase "a function returns a pointer" really means "a function returns an address."

If you are ever in doubt as to what is really contained in a variable or how it should be treated, use the `printf( )` function to display the contents of the variable, the "thing pointed to," or "the address of the variable." Seeing what is displayed frequently helps sort out what is really in the variable.

10.8 Chapter Summary

1. All variables have an lvalue and an rvalue. The *lvalue* is the address of the variable and the *rvalue* is the contents of the variable. Programmers typically use a variable's name to reference the variable's contents (its rvalue), while computers typically use a variable's name to reference its address (lvalue). The address operator, `&`, can be used to obtain a variable's lvalue.
2. A pointer is a variable that is used to store the address of another variable. Pointers, like all C variables, must be declared. The indirection operator, `*`, is used both to declare a pointer variable and to access the variable whose address is stored in a pointer.
3. An array name is a pointer constant. The value of the pointer constant is the address of the first element in the array. Thus, if `val` is the name of an array, `val` and `&val[0]` can be used interchangeably.
4. Any reference to an array element using subscript notation can always be replaced using pointer notation. That is, the notation `a[i]` can always be replaced by the notation `*(a + i)`. This is true whether `a` was initially declared explicitly as an array or as a pointer.
5. Arrays are passed to functions by reference. The called function always receives direct access to the originally declared array elements.
6. When a single-dimensional array is passed to a function, the argument

declaration for the function can be either an array declaration or a pointer declaration. Thus, the following argument declarations are equivalent:

```
float a[ ];

float *a;
```

7. Pointer variables can be incremented, decremented, and compared. Numbers added to or subtracted from a pointer are automatically scaled. The scale factor used is the number of bytes required to store the data type originally pointed to.

Character Strings

Chapter Eleven

11.1 String Fundamentals
11.2 Pointers and Library Functions
11.3 String Definitions and Pointer Arrays
11.4 Formatting Strings
11.5 Common Programming Errors
11.6 Chapter Summary

On a fundamental level, strings are simply arrays of characters that can be manipulated using standard element-by-element array-processing techniques. On a higher level, string library functions are available for treating strings as complete entities. This chapter explores the input, manipulation, and output of strings using both approaches. We will also examine the close connection between string-handling functions and pointers.

11.1 String Fundamentals

A string constant, informally referred to as a string, is any sequence of characters enclosed in double quotes. For example, `"This is a string"`, `"Hello World!"`, and `"xyz 123 *!#@&"` are all strings.

A string is stored as an array of characters terminated by a special end-of-string marker called the null character. The null character, represented by the escape sequence `\0`, is the sentinel marking the end of the string. For example, Figure 11–1 illustrates how the string `"Good Morning!"` is stored in memory. The string uses fourteen storage locations, with the last character in the string being the end-of-string marker `\0`. The double quotes are not stored as part of the string.

Since a string is stored as an array of characters, the individual characters in the array can be input, manipulated, or output using standard array-handling techniques utilizing either subscript or pointer notations. The end-of-string null character is useful as a sentinel for detecting the end of the string.

String Input and Output

Although the programmer has the choice of using either a library or a user-written function for processing a string already in memory, inputting a string from a keyboard or displaying a string always requires some reliance on standard library functions. Table 11–1 lists the commonly available library functions for both character-by-character and complete string input/output.

FIGURE 11–1 Storing a String in Memory

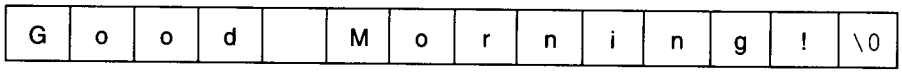


TABLE 11–1 Standard String or Character Library Functions

Input	Output
`gets( )`	`puts( )`
`scanf( )`	`printf( )`
`getchar( )`	`putchar( )`

The gets( ) and puts( ) functions deal with strings as complete units. Both are written using the more elemental routines getchar( ) and putchar( ). The getchar( ) and putchar( ) routines provide for the input and output of individual characters. Programs that access any of these four routines must contain the stdio.h header file, which contains definitions required by the accessed library functions.

Program 11-1 illustrates the use of gets( ) and puts( ) to input and output a string entered at the user's terminal.

Program 11-1

```
#include <stdio.h>
main( )
{
  char message[81];    /* enough storage for a complete line */

  printf("Enter a string:\n");
  gets(message);
  printf("The string just entered is:\n");
  puts(message);
}
```

The following is a sample run of Program 11-1:

```
Enter a string:
This is a test input of a string of characters.
The string just entered is:
This is a test input of a string of characters.
```

The gets( ) function used in Program 11-1 continuously accepts and stores the characters typed at the terminal into the character array named message. Pressing the ENTER key at the terminal generates a newline character, \n, which is interpreted by gets( ) as the end-of-character entry. All the characters encountered by gets( ), except the newline character, are stored in the message array. Before returning, the gets( ) function appends the null character to the stored set of characters, as illustrated in Figure 11–2a. The puts( ) function is then used to display the string. As illustrated in Figure 11–2b, the puts( ) function automatically sends a newline escape sequence to the display terminal after the string has been printed.

In general, a printf( ) function call can always be used in place of a puts( ) function call. For example, the statement printf("%s\n",message); is a direct replacement for the statement puts(message); used in Program 11-1. The newline escape sequence in the printf( ) function call substitutes for the automatic newline generated by puts( ) after the string is displayed.

The one-to-one correspondence between the output functions printf( ) and puts( ) is not duplicated by the input functions scanf( ) and gets( ). For

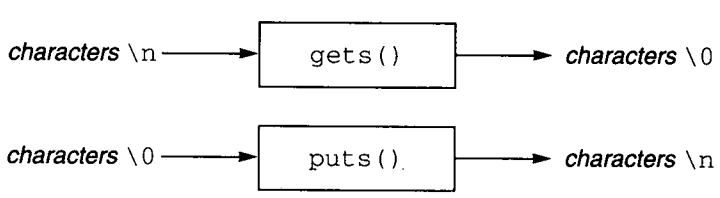


FIGURE 11–2 Inputting and Outputting a String Using the `gets( )` and `puts( )` Functions

example, `scanf("%s",message)` and `gets(message)` are not equivalent. The `scanf( )` function reads a set of characters up to either a blank space or a newline character, whereas `gets( )` stops accepting characters only when a newline is detected. Trying to enter the characters `This is a string` using the statement `scanf("%s",message);` results in the word `This` being assigned to the `message` array. Entering the complete line using a `scanf( )` function call would require a statement such as:

```
scanf("%s %s %s %s", message1, message2, message3, message4);
```

Here, the word `This` would be assigned to the string `message1`, the word `is` assigned to the string `message2`, and so on. The fact that a blank is used as a delimiter by `scanf( )` means that this function isn't that useful for entering string data.

Note that if the `scanf( )` function is used for inputting string data, the `&` is not used before the array name. Since an array name is a pointer constant equivalent to the address of the first storage location reserved for the array, `message` is the same as `&message[0]`. Thus, the function call `scanf("%s",&message[0])` can be replaced by `scanf("%s",message)`.

String Processing

Strings can be manipulated using either standard library functions or standard array-processing techniques. The library functions typically available for use are presented in the next section. For now we will concentrate on processing a string in a character-by-character fashion. This will allow us to understand how the standard library functions are constructed and to create our own library functions. For a specific example, consider the function `strcopy( )` that copies the contents of `string2` to `string1`.

```
strcopy(char string1[], char string2[])   /* copy string2 to string1 */
{
  int i = 0;                    /* i will be used as a subscript */

  while ( string2[i] != '\0') /* check for the end-of-string  */
  {
    string1[i] = string2[i];  /* copy the element to string1 */
    ++i;
  }
  string1[i] = '\0';          /* terminate the first string */
  return;
}
```

Although this string copy function can be shortened considerably and written more compactly, the function illustrates the main features of string manipulation. The two strings are passed to `strcopy1` as arrays. Each element of `string2` is then assigned to the equivalent element of `string1` until the end-of-string marker is encountered. The detection of the null character forces the termination of the `while` loop controlling the copying of elements. Since the null character is not copied from `string2` to `string1`, the last statement in `strcopy( )` appends an end-of-string character to `string1`. Prior to calling `strcopy( )`, the programmer must ensure that sufficient space has been allocated for the `string1` array to accomodate the elements of the `string2` array.

Program 11-2 includes the `strcopy( )` function in a complete program. Notice that the function prototype for `strcopy( )` in `main( )` declares that the function expects to receive the addresses of the beginnings of two character arrays.

Program 11-2

```
#include <stdio.h>
main( )
{
  char message[81];   /* enough storage for a complete line   */
  char new_mess[81];  /* enough storage for a copy of message */
  int i;
  void strcopy(char [ ], char [ ]); /* function prototype */

  printf("Enter a sentence: ");
  gets(message);
  strcopy(new_mess,message);   /* pass two array addresses */
  puts(new_mess);

}

void strcopy(char string1[], char string2[])   /* copy string2 to string1 */
{
  int i = 0;                  /* i will be used as a subscript */

  while ( string2[i] != '\0') /* check for the end-of-string  */
  {
    string1[i] = string2[i];  /* copy the element to string1 */
    ++i;
  }
  string1[i] = '\0';          /* terminate the first string */
  return;
}
```

The following is a sample run of Program 11-2:

```
Enter a sentence: How much wood could a woodchuck chuck.
How much wood could a woodchuck chuck.
```

Character-by-Character Input

Just as strings can be processed using character-by-character techniques, they can be entered and displayed in this manner. For example, consider Program 11-3, which uses the character-input function `getchar( )` to construct a string one character at a time. The boxed portion of Program 11-3 essentially replaces the `gets( )` function previously used in Program 11-1.

Program 11-3

```
#include <stdio.h>
main( )
{
  char message[81],c;    /* enough storage for a complete line */
  int i;

  printf("Enter a sentence:\n");

  i = 0;
  while( i < 81 && (c = getchar( )) != '\n')
  {
    message[i] = c;       /* store the character entered */
    ++i;
  }
  message[i] = '\0';      /* terminate the string */

  printf("The sentence just entered is:\n");
  puts(message);
}
```

The following is a sample run of Program 11-3:

```
Enter a string:
This is a test input of a string of characters.
The string just entered is:
This is a test input of a string of characters.
```

The `while` statement in Program 11-3 causes characters to be read providing the number of characters entered is less than 81 and the character returned by `getchar( )` is not the newline character. The parentheses around the expression `c = getchar( )` are necessary to assign the character returned by `getchar( )` to the variable `c` prior to comparing it to the newline escape sequence. Otherwise, the comparison operator, `!=`, which takes precedence over the assignment operator, causes the entire expression to be equivalent to

```
c = (getchar() != '\n')
```

This has the effect of first comparing the character returned by `getchar` to `'\n'`. The value of the relational expression `getchar( ) != '\n'` is either `0` or `1`, depending on whether or not `getchar( )` received the newline character. The value assigned to `c` then would also be either `0` or `1`, as determined by the comparison.

Program 11-3 also illustrates a very useful technique for developing functions. The boxed statements constitute a self-contained unit for entering a complete line of characters from a terminal. As such, these statements can be removed from `main( )` and placed together as a new function. Program 11-4 illustrates placing these statements in a new function called `getline( )`.

Program 11-4

```
#include <stdio.h>
main( )
{
  char message[81];   /* enough storage for a complete line */
  int i;
  void getline(char [ ]);  /* function prototype */

  printf("Enter a string:\n");
  getline(message);
  printf("The string just entered is:\n");
  puts(message);
}

void getline(char strng[ ])
{
  int i = 0;
  char c;
  while( i < 81 && (c = getchar( )) != '\n')
  {
    strng[i] = c;       /* store the character entered */
    ++i;
  }
  strng[i] = '\0';       /* terminate the string       */
  return;
}
```

We can go further with `getline( )` and write it more compactly by having the character returned by `getchar( )` assigned directly to the `strng` array. This eliminates the need for the local variable `c` and results in the following version:

```
getline(char strng[ ])
{
  int i = 0;
  while( i < 81 && (strng[i++] = getchar( )) != '/n')
    ;
  strng[i] = '\0';       /* terminate the string */
  return;
}
```

Notice that in addition to assigning the returned character from `getchar( )` directly to the `strng` array, the assignment statement

```
strng[i++] = getchar( )
```

additionally increments the subscript `i` using the postfix operator, `++`. The null statement, `;`, then fulfills the requirement that a `while` loop contain at least one statement. Both versions of `getline( )` are suitable replacements for `gets( )`, and show the interchangeability between user-written and library functions.

C's enormous flexibility is shown by this ability to replace a library function with a user-written version and its ability to have functions written in various ways. Neither version of `getline( )` is "more correct" from a programming standpoint. Each version presented (and more versions can be created) has its advantages and disadvantages. While the second version is more compact, the first version is clearer to beginning programmers. In creating your own C programs, select a style that is comfortable and remain with it until your growing programming expertise dictates modifications to your style.

Exercises 11.1

1a. The following function can be used to select and display all vowels contained within a user-input string:

```
vowels(char strng[ ])
{
  int i = 0;
  char c;

  while ((c = strng[i++]) != '\0')
    switch(c)
    {
       case 'a':
       case 'e':
       case 'i':
       case 'o':
       case 'u':
         putchar(c);
    } /* end of switch */
    putchar('\n');
}
```

Notice that the `switch` statement in `vowels( )` uses the fact that selected cases "drop through" in the absence of `break` statements. Thus, all selected cases result in a `putchar( )` function call. Include `vowels( )` in a working program that accepts a user-input string and then displays all vowels in the string. In response to the input `How much is the little worth worth?`, your program should display `ouieieoo`.

b. Modify `vowels( )` to count and display the total number of vowels contained in the string passed to it.

2. Modify the `vowels( )` function given in Exercise 1a to count and display the individual numbers of each vowel contained in the string.

3.a. Write a C function to count the total number of characters, including blanks, contained in a string. Do not include the end-of-string marker in the count.

b. Include the function written for Exercise 3a in a complete working program.

4. Write a program that accepts a string of characters from a terminal and displays the hexadecimal equivalent of each character.

5. Write a C program that accepts a string of characters from a terminal and displays the string one word per line.

6. Write a function that reverses the characters in a string. (*Hint:* This can be considered as a string copy starting from the back end of the first string.)

7. Write a function called `del_char( )` that can be used to delete characters from a string. The function should take three arguments: the string name, the number of characters to delete, and the starting position in the string where characters should be deleted. For example, the function call `del_char(strng,13,5)`, when applied to the string `all enthusiastic people`, should result in the string `all people`.

8. Write a function call `add_char( )` to insert one string of characters into another string. The function should take three arguments: the string to be inserted, the original string, and the position in the original string where the insertion should begin. For example, the call `add_char("for all",message,6)` should insert the characters `for all` in `message` starting at `message[5]`.

9a. Write a C function named `to_upper( )` that converts lowercase letters into uppercase letters. The expression `c - 'a' + 'A'` can be used to make the conversion for any lowercase character stored in c.

b. Add a data input check to the function written in Exercise 9a to verify that a valid lowercase letter is passed to the function. A character is lowercase if it is greater than or equal to `a` and less than or equal to `z`. If the character is not a valid lowercase letter, have the function `to_upper( )` return the passed character unaltered.

c. Write a C program that accepts a string from a terminal and converts all lowercase letters in the string to uppercase letters.

10. Write a C program that counts the number of words in a string. A word is encountered whenever a transition from a blank space to a nonblank character is encountered. Assume the string contains only words separated by blank spaces.

11.2 Pointers and Library Functions

Pointers are exceptionally useful in constructing string-handling functions. When pointer notation is used in place of subscripts to access individual characters in a string, the resulting statements are both more compact and more efficient. In this section we describe the equivalence between subscripts and pointers when accessing individual characters in a string.

Consider the `strcopy( )` function introduced in the previous section. This function was used to copy the characters of one string to a second string. For convenience, this function is repeated below:

```
strcopy(char string1[ ], char string2[ ])   /* copy string2 to string1 */
{
  int i = 0;

  while ( string2[i] != '\0')   /* check for the end-of-string */
  {
    string1[i] = string2[i];    /* copy the element to string1 */
    ++i;
  }
  string1[i] = '\0';            /* terminate the first string */
  return;
}
```

The function `strcopy( )` is used to copy the characters from one array to another array, one character at a time. As currently written, the subscript `i` in the function is used successively to reference each character in the array named `string2` by "marching along" the string one character at a time. Before we write a pointer version of `strcopy( )`, we will make two modifications to the function to make it more efficient.

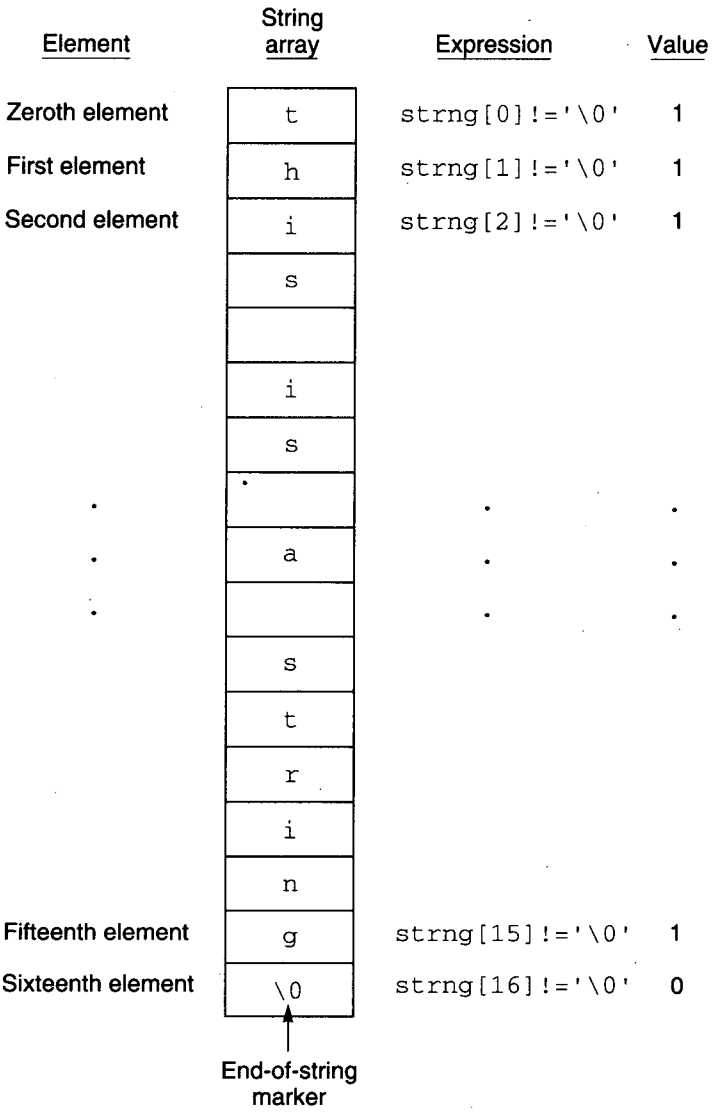
The `while` statement in `strcopy( )` tests each character to ensure that the end of the string has not been reached. As with all relational expressions, the tested expression, `string[i] != '\0'`, is either true or false. Using the string `this is a string` illustrated in Figure 11–3 as an example, as long as `string[i]` does not reference the end-of-string character the value of the expression is nonzero and is considered to be true. The expression is only false when the value of the expression is zero. This occurs when the last element in the string is accessed.

Recall that C defines false as zero and true as anything else. Thus, the expression `string[i] != '\0'` becomes zero, or false, when the end of the string is reached. It is nonzero, or true, everywhere else. Since the null character has an internal value of zero by itself, the comparison to `'\0'` is not necessary. When `string[i]` references the end-of-string character, the value of `string[i]` is zero. When `string[i]` references any other character, the value of `string[i]` is the value of the code used to store the character and is nonzero. Figure 11–4 lists the ASCII codes for the string `this is a string`. As seen in the figure, each element has a nonzero value except for the null character.

Since the expression `string[i]` is only zero at the end of a string and nonzero for every other character, the expression `while (strng[i] != '\0')` can be replaced by the simpler expression `while (strng[i])`. Although this may appear confusing at first, the revised test expression is certainly more compact than the longer version. Since end-of-string tests are frequently written by advanced C programmers in this shorter form, it is worthwhile being familiar with this expression. Including this expression in `strcopy( )` results in the following version:

```
strcopy(char string1[ ], char string2[ ])   /* copy string2 to string1 */
{
  int i = 0;

  while (string2[i])
  {
    string1[i] = string2[i];     /* copy the element to string1 */
    ++i;
  }
  string1[i] = '\0';             /* terminate the first string */
  return;
}
```

FIGURE 11–3 The `while` Test Becomes False at the End of the String



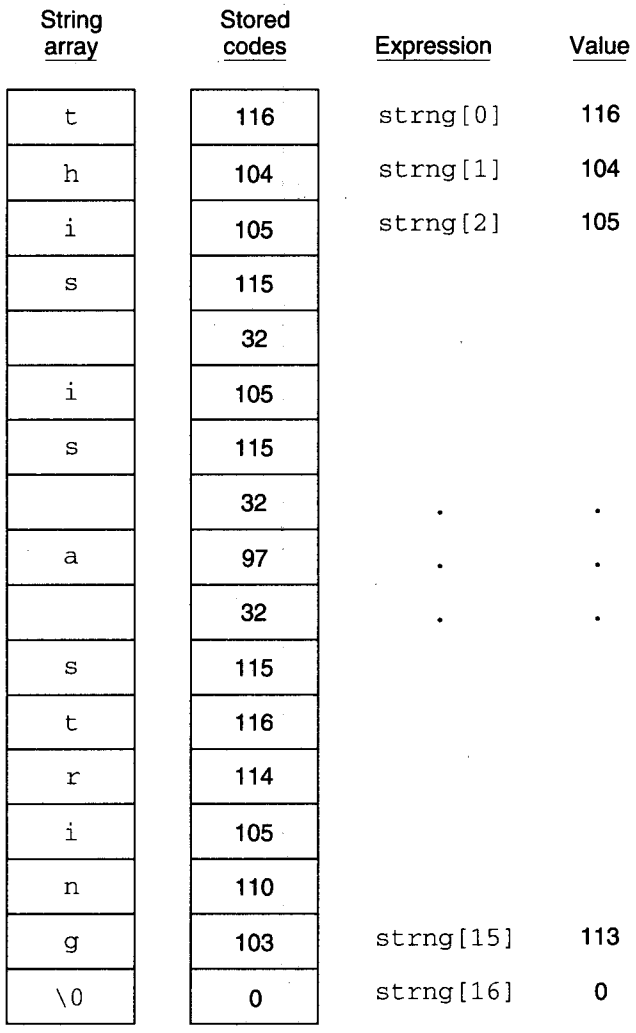


FIGURE 11–4 The ASCII Codes Used to Store `this is a string`

The second modification that can be made to this string copy function is to include the assignment inside the test portion of the `while` statement. Our new version of the string copy function is:

```
strcopy(char string1[], char string2[])   /* copy string2 to string1 */
{
  int i = 0;

  while (string1[i] = string2[i])
    ++i;
  return;
}
```

Notice that including the assignment statement within the test part of the `while` statement eliminates the necessity of separately terminating the first string with the null character. The assignment within the parentheses ensures that the null character is copied from the second string to the first string. The

value of the assignment expression only becomes zero after the null character is assigned to `string1`, at which point the `while` loop is terminated.

The conversion of `strcopy( )` from subscript notation to pointer notation is now straightforward. Although each subscript version of `strcopy` can be rewritten using pointer notation, the following is the equivalent of our last subscript version:

```
strcopy(char *string1, char *string2)   /* copy string2 to string1 */
{
  while (*string1 = *string2)
  {
    string1++;
    string2++;
  }
  return;
}
```

In both subscript and pointer versions of `strcopy( )`, the function receives the name of the array being passed. Recall that passing an array name to a function actually passes the address of the first location of the array. In our pointer version of `strcopy( )` the two passed addresses are stored in the pointer arguments `string2` and `string1`, respectively.

The declarations `char *string1;` and `char *string2;` used in the pointer version of `strcopy( )` indicate that `string1` and `string2` are both pointers containing the address of a character, and stress the treatment of the passed addresses as pointer values rather than array names. These declarations are equivalent to the declarations `char string1[ ]` and `char string2[ ]`, respectively.

Internal to `strcopy( )`, the pointer expression `*string1`, which refers to "the element whose address is in `string1`," replaces the equivalent subscript expression `string1[i]`. Similarly, the pointer expression `*string2` replaces the equivalent subscript expression `string2[i]`. The expression `*string1 = *string2` causes the element pointed to by `string2` to be assigned to the element pointed to by `string1`. Since the starting addresses of both strings are passed to `strcopy( )` and stored in `string1` and `string2`, respectively, the expression `*string1` initially refers to `string1[0]` and the expression `*string2` initially refers to `string2[0]`.

Consecutively incrementing both pointers in `strcopy( )` with the expressions `string1++` and `string2++` simply causes each pointer to "point to" the next consecutive character in the respective string. As with the subscript version, the pointer version of `strcopy` steps along, copying element by element, until the end of the string is copied.

One final change to the string copy function can be made by including the pointer increments as postfix operators within the test part of the `while` statement. The final form of the string copy function is:

```
strcopy(char *string1, char *string2)   /* copy string2 to string1 */
{
  while ( *string1++ = *string2++ )
    ;
  return;
}
```

There is no ambiguity in the expression `*string1++ = *string2++` even though the indirection operator, *, and the increment operator, ++, have the same precedence. These operators associate from left to right, so the character pointed to is accessed before the pointer is incremented. Only after completion of the assignment `*string1 = *string2` are the pointers incremented to correctly point to the next characters in the respective strings.

Most C compilers include a string copy function in their standard library. This library function is typically written exactly like our pointer version of `strcopy( )`.

Library Functions

Extensive collections of string-handling functions and routines are included with most C compilers. These were previously listed in Section 7.1, and for convenience are repeated in Table 11–2.

Library functions and routines are called in the same manner that all C functions are called. This means that if a library function returns a value the function must be declared within your program before it is called. For example, if a library function named `strngfoo( )` returns a pointer to a character, the calling

TABLE 11–2 String and Character Library Routines

Name	Description
`strcat(string1,string2)`	Concatenates `string2` to `string1`.
`strchr(string,character)`	Locates the position of the first occurence of the character within the string. Returns the address of the character.
`strcmp(string1,string2)`	Compares `string2` to `string1`.
`strcpy(string1,string2)`	Copies `string2` to `string1`.
`strlen(string)`	Returns the length of the string.
`isalpha(character)`	Returns a nonzero number if the character is a letter; otherwise it returns a zero.
`isupper(character)`	Returns a nonzero number if the character is uppercase; otherwise it returns a zero.
`islower(character)`	Returns a nonzero number if the character is lowercase; otherwise it returns a zero.
`isdigit(character)`	Returns a nonzero number if the character is a digit (0 through 9); otherwise it returns a zero.
`toupper(character)`	Returns the uppercase equivalent if the character is lowercase; otherwise it returns the character unchanged.
`tolower(character)`	Returns the lowercase equivalent if the character is uppercase; otherwise it returns the character unchanged.

function must be alerted that an address is being returned. Thus, the statement `char *strngfoo( );`, which declares that `strngfoo( )` returns the address of a character (pointer to `char`), must be placed either as an external declaration or directly within the calling function's variable declarations. Alternatively, the string header file `<string.h>` may be included in place of individual string function prototypes.

Before attempting to use any standard library functions, check that they are included in the C compiler available on your computer system. Be careful to check the type of arguments expected by the function, the data type of any returned value, and whether any standard files, such as `<string.h>` or `<ctype.h>`, need be included in your program to access these routines.

Exercises 11.2

1. Determine the value of `*text`, `*(text + 3)`, and `*(text + 10)`, assuming that `text` is an array of characters and the following has been stored in the array:

a. now is the time
b. rocky raccoon welcomes you
c. Happy Holidays
d. The good ship

2a. The following function, `convert( )`, "marches along" the string passed to it and sends each character in the string one at a time to the `to_upper( )` function until the null character is encountered:

```
convert(char strng[ ])        /* convert a string to uppercase letters */
{
  int i = o;

  while (strng[i] != '\0')
  {
    strng[i] = to_upper(strng[i]);
    ++i;
  }
  return;
}

to_upper(letter)  /* convert a character to uppercase */
char letter;
{
  if( letter >= 'a' && letter <= 'z')
    return (letter - 'a' + 'A');
  else
    return (letter);
}
```

The `to_upper( )` function takes each character passed to it and first examines it to determine if the character is a lowercase letter (a lowercase letter is any character between a and z, inclusive). Assuming that characters are stored using the standard ASCII character codes, the expression `letter - 'a' + 'A'` converts a lowercase letter to its uppercase equivalent. Rewrite the `convert( )` function using pointers.

b. Include the `convert( )` and `to_upper( )` functions in a working program. The

program should prompt the user for a string and echo the string back to the user in uppercase letters. Use `gets( )` and `puts( )` for string input and display.

3. Using pointers, repeat Exercise 1 from Section 11.1.

4. Using pointers, repeat Exercise 2 from Section 11.1.

5. Using pointers, repeat Exercise 3 from Section 11.1.

6. Write a function named `remove( )` that deletes all occurrences of a character from a string. The function should take two arguments: the string name and the character to be removed. For example, if `message` contains the string `Happy Holidays`, the function call `remove(message,'H')` should place the string `appy olidays` into `message`.

7. Using pointers, repeat Exercise 6 from Section 11.1.

8. Write a program using the `getchar( )`, `toupper( )`, and `putchar( )` library functions to echo back each letter entered in its uppercase form. The program should terminate when the digit 1 key is pressed.

9. Write a function that uses pointers to add a single character at the end of an existing string. The function should replace the existing `\0` character with the new character and append a new `\0` at the end of the string.

10. Write a function that uses pointers to delete a single character from the end of a string. This is effectively achieved by moving the `\0` character one position closer to the start of the string.

11. Determine the string-handling functions that are available with your C compiler. For each available function list the data types of the arguments expected by the function and the data type of any returned value.

12. Write a function named `trimfrnt( )` that deletes all leading blanks from a string. Write the function using pointers.

13. Write a function named `trimrear( )` that deletes all trailing blanks from a string. Write the function using pointers.

14. Write a function named `strlen( )` that returns the number of characters in a string. Do not include the `\0` character in the returned count.

11.3 String Definitions and Pointer Arrays

The definition of a string automatically involves a pointer. For example, the definition `char message1[81];` both reserves storage for 81 characters and automatically creates a pointer constant, `message1`, which contains the address of `message1[0]`. As a pointer constant, the address associated with the pointer cannot be changed—it must always "point to" the beginning of the created array.

Instead of initially creating a string as an array, however, it is also possible to create a string using a pointer. This is similar in concept to declaring a passed array as either an array or a pointer argument internal to the receiving function. For example, the definition `char *message2;` creates a pointer to a character. In this case, `message2` is a true pointer variable. Once a pointer to a character is defined, assignment statements, such as `message2 ="this is a string";`, can be made. In this assignment, `message2` receives the address of the first location used by the computer to store the string.

The main difference in the definitions of `message1` as an array and `message2` as a pointer is the way the pointer is created. Defining `message1` using the declaration `char message1[81]` explicitly calls for a fixed amount of stor-

age for the array. This causes the compiler to create a pointer constant. Defining `message2` using the declaration `char *message2` explicitly creates a pointer variable first. This pointer is then used to hold the address of a string when the string is actually specified. This difference in definitions has both storage and programming consequences.

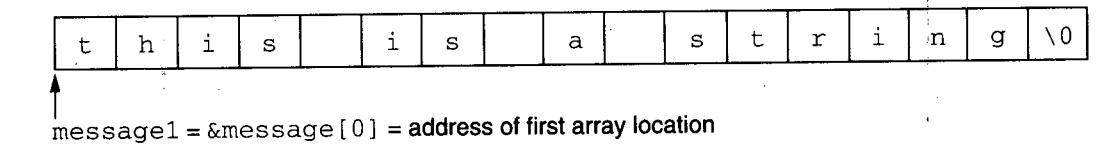
From a programming perspective, defining `message2` as a pointer to a character allows string assignments, such as `message2 = "this is a string";`, to be made. Similar assignments are not allowed for strings defined as arrays. Thus, the statement `message1 = "this is a string";` is not valid. Both definitions, however, allow initializations to be made using a string assignment. For example, both of the following initializations are valid:

```
char message1[ ] = "this is a string";
char *message2 = "this is a string";
```

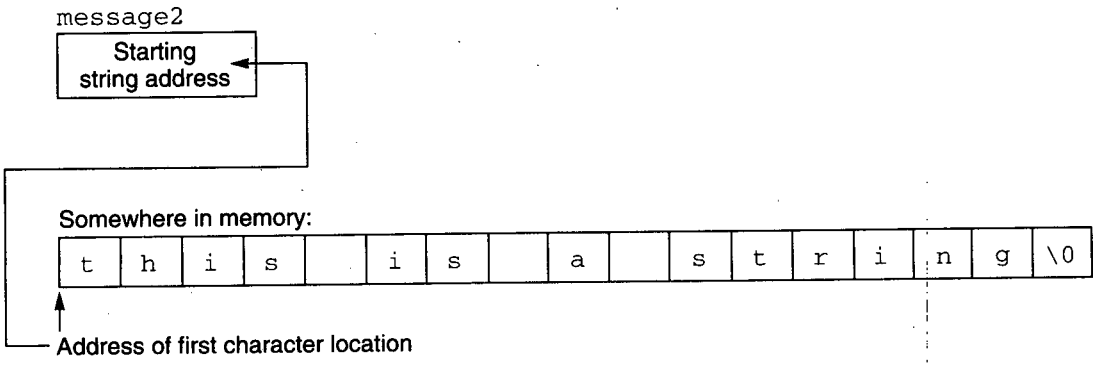
From a storage perspective, however, the allocation of space for `message1` and `message2` is different, as illustrated in Figure 11–5. As shown in the figure, both initializations cause the computer to store the same string internally. In the case of `message1`, a specific set of 81 storage locations is reserved and the first 17 locations are initialized. For `message1`, different strings can be stored, but each string will overwrite the previously stored characters. The same is not true for `message2`.

The definition of `message2` reserves enough storage for one pointer. The initialization then causes the string to be stored and the starting storage address of

FIGURE 11–5 String Storage Allocation



a. Storage allocation for a string defined as an array



b. Storage of a string using a pointer

the string to be loaded into the pointer. If a later assignment is made to message2, the initial string remains in memory and new storage locations are allocated to the new string. Program 11-5 uses the message2 character pointer to successively "point to" two different strings.

Program 11-5

```
#include <stdio.h>
main( )
{
  char *message2 = "this is a string";

  printf("\nThe string is: %s", message2);
  printf("\n The first address of this string is %p", message2);

  message2 = "A new message";
  printf("\nThe string is now: %s", message2);
  printf("\n The first address of this string is %p", message2);
}
```

A sample output for Program 11-5 is:[1]

```
The string is: this is a string
 The first address of this string is 009E
The string is now: A new message
 The first address of this string is 00EB
```

In Program 11-5, the variable message2 is initially created as a pointer variable and loaded with the starting storage address of the first string. The printf( ) function is then used to display this string. When the %s conversion control sequence is encountered by printf( ), it alerts the function that a string is being referenced. The printf( ) function then expects either a string constant or a pointer containing the address of the first character in the string. This pointer can be either an array name or a pointer variable. The printf( ) function uses the address provided to correctly locate the string, and then continues accessing and displaying characters until it encounters a null character. As illustrated by the output, the hexadecimal address of the first character in the string is 009E.

After the first string and its starting address is displayed, the next assignment statement in Program 11-5 causes the computer to store a second string and change the address in message2 to point to the starting location of this new string. The printf( ) function then displays this string and its starting storage address.

[1] The actual addresses used by this program for the storage of messages is machine dependent.

It is important to realize that the second string assigned to `message2` does not overwrite the first string, but simply changes the address in `message2` to point to the new string. As illustrated in Figure 11–6, both strings are stored inside the computer. Any additional string assignment to `message2` would result in the additional storage of the new string and a corresponding change in the address stored in `message2`.

Pointer Arrays

The declaration of an array of character pointers is an extremely useful extension to single string pointer declarations. For example, the declaration

```
char *seasons[4];
```

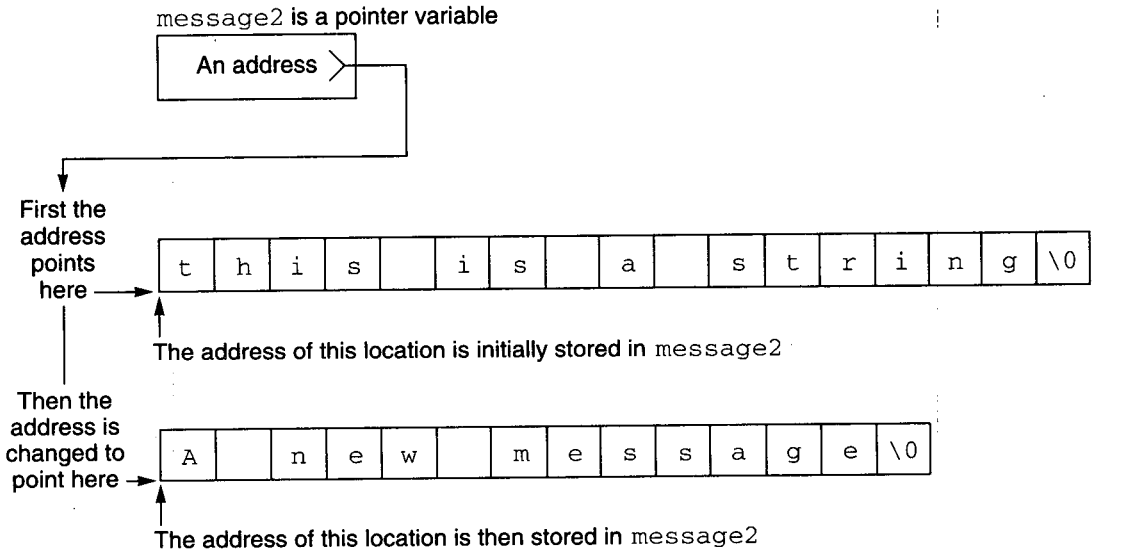
creates an array of four elements, where each element is a pointer to a character. As individual pointers, each pointer can be assigned to point to a string using string assignment statements. Thus, the statements

```
seasons[0] = "Winter";
seasons[1] = "Spring";
seasons[2] = "Summer";
seasons[3] = "Fall";
```

set appropriate addresses into the respective pointers. Figure 11–7 illustrates the addresses loaded into the pointers for these assignments.

As illustrated in Figure 11–7, the `seasons` array does not contain the actual strings assigned to the pointers. These strings are stored elsewhere in the computer, in the normal data area allocated to the program. The array of pointers contains only the addresses of the starting location for each string.

FIGURE 11–6 Storage Allocation for Program 11-5



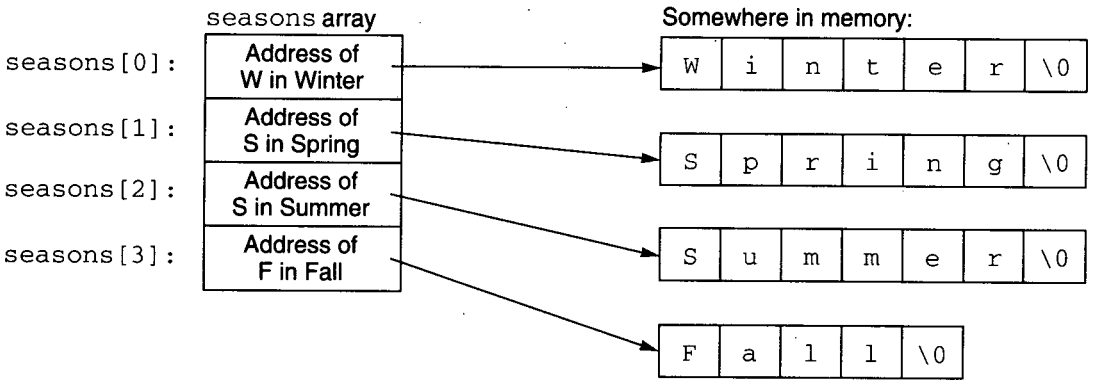


FIGURE 11–7 The Addresses Contained in the `seasons[ ]` Pointers

The initializations of the `seasons` array can also be incorporated directly within the definition of the array, as follows:

```
char *seasons[4] = { "Winter",
                     "Spring",
                     "Summer",
                     "Fall"};
```

This declaration both creates an array of pointers and initializes the pointers with appropriate addresses. Once addresses have been assigned to the pointers, each pointer can be used to access its corresponding string. Program 11-6 uses the `seasons` array to display each season using a `for` loop.

Program 11-6

```
#include <stdio.h>
main( )
{
  int n;
  char *seasons[ ] = { "Winter",
                       "Spring",
                       "Summer",
                       "Fall"};

  for( n = 0; n < 4; ++n)
  printf("\nThe season is %s.",seasons[n]);
}
```

The output obtained for Program 11-6 is:

```
The season is Winter.
The season is Spring.
The season is Summer.
The season is Fall.
```

The advantage of using a list of pointers is that logical groups of data headings can be collected together and accessed with one array name. For example, the months in a year can be collectively grouped in one array called `months`, and the days in a week collectively grouped together in an array called `days`. The grouping of like headings allows the programmer to access and print an appropriate heading by simply specifying the correct position of the heading in the array. Program 11-7 uses the `seasons` array to correctly identify and display the season corresponding to a user-input month.

Program 11-7

```
#include <stdio.h>
main( )
{
  int n;
  char *seasons[ ] = { "Winter",
                       "Spring",
                       "Summer",
                       "Fall"};

  printf("\nEnter a month (use 1 for Jan., 2 for Feb., etc.): ");
  scanf("%d", &n);
  n = (n % 12) / 3;     /* create the correct subscript */
  printf("The month entered is a %s month.",seasons[n]);
}
```

Except for the expression `n = (n % 12) / 3`, Program 11-7 is rather straightforward. The program requests the user to input a month and accepts the number corresponding to the month using a `scanf( )` function call.

The expression `n = (n % 12) / 3` uses a common program "trick" to scale a set of numbers into a more useful set. Using subscripts, the four elements of the `seasons` array must be accessed using a subscript from 0 through 3. Thus, the months of the year, which correspond to the numbers 1 through 12, must be adjusted to correspond to the correct season subscript. This is done using the expression `n = (n % 12) / 3`. The expression `n % 12` adjusts the month entered to lie within the range 0 through 11, with 0 corresponding to December, 1 for January, and so on. Dividing by 3 causes the resulting number to range between 0 and 3, corresponding to the possible seasons elements. The result of the divi-

sion by 3 is assigned to the integer variable n. The months 0, 1, and 2, when divided by 3, are set to 0; the months 3, 4, and 5 are set to 1; the months 6, 7, and 8 are set to 2; and the months 9, 10, and 11 are set to 3. This is equivalent to the following assignments:

Months	Season
December, January, February	Winter
March, April, May	Spring
June, July, August	Summer
September, October, November	Fall

The following is a sample output obtained for Program 11-7:

```
Enter a month (use 1 for Jan., 2 for Feb., etc.): 12
The month entered is a Winter month.
```

Exercises 11.3

1. Write two declaration statements that can be used in place of the declaration `char text[ ] = "Hooray!";`.

2. Determine the value of `*text`, `*(text + 3)`, and `*(text + 7)` for each of the following sections of code:

a.
```
char *text;
char message[ ] = "the check is in the mail";
text = message;
```

b.
```
char *text;
char formal[ ] = {'t','h','i','s',' ','i','s',' ','a','n',
                  'i','n','v','i','t','a','t','i','o','n','\0'};
text = &formal[0];
```

c.
```
char *test;
char more[ ] = "Happy Holidays";
text = &more[4];
```

d.
```
char *text, *second;
char blip[ ] = "The good ship";
second = blip;
text = ++second;
```

3. Determine the error in the following program:

```
#include <stdio.h>
main( )
{
  int i = 0;
  char message[ ] = {'H','e','l','l','o','\0'};

  for( ; i < 5; ++i)
```

```
{
  putchar(*message);
  ++message:
}
```

4 *a.* Write a C function that displays the day of the week corresponding to a user-entered input number between 1 and 7. For example, in response to an input of 2, the program displays the name Monday. Use an array of pointers in the function.
b. Include the function written for Exercise 4a in a complete working program.

5. Modify the function written in Exercise 4a so that the function returns the address of the character string containing the proper month to be displayed.

6. Write a C function that will accept ten lines of user-input text and store the entered lines as ten individual strings. Use a pointer array in your function.

11.4 Formatting Strings

Besides the special string-handling functions in the standard library provided with your C compiler, both the `printf( )` and `scanf( )` functions have string-formatting capabilities. Additionally, two related functions, `sprintf( )` and `sscanf( )`, provide further string-processing features. In this section we present the additional features that these functions provide when used with strings.

Field width specifiers can be included in a `printf( )` conversion control sequence. These specifiers can also be used with the `%s` conversion control sequence to control the display of a string. For example, the statement

```
printf("|%25s|","Have a Happy Day");
```

displays the message `Have a Happy Day`, right justified in a field of 25 characters, as follows:

```
|                    Have a Happy Day|
```

We have placed a bar (`|`) at the beginning and end of the string field to clearly delineate the field being printed. Placing a minus sign (`-`) in front of the field width specifier forces the string to be left justified in the field. For example, the statement

```
printf("|%-25s|","Have a Happy Day");
```

causes the display:

```
|Have a Happy Day           |
```

If the field width specifier is too small for the string, the specifier is ignored and the string is displayed using sufficient space to accommodate the complete string.

The precision specifier used for determining the number of digits displayed to the right of a decimal number can also be used as a string specifier. When used

with strings, the precision specifier determines the maximum number of characters that will be displayed. For example, the statement

```
printf("|%25.12s|","Have a Happy Day");
```

causes the first 12 characters in the string to be displayed, right justified, in a field of 25 characters. This produces the display:

```
|                    Have a Happy|
```

Similarly, the statement

```
printf("|%-25.12s|","Have a Happy Day");
```

causes 12 characters to be left justified in a field of 25 characters. This produces the display:

```
|Have a Happy               |
```

When a precision specifier is used with no field width specifier, the indicated number of characters is displayed in a field sufficiently large to hold the designated number of characters. Thus, the statement

```
printf("|%.12s|","Have a Happy Day");
```

causes the first 12 characters in the string to be displayed in a field of 12 characters. If the string has less than the number of characters designated by the precision specifier, the display is terminated when the end-of-string is encountered.

In-Memory String Conversions

While `printf( )` displays data to the standard device used by your computer for output and `scanf( )` scans the standard device used for input, the `sprintf( )` and `sscanf( )` functions provide similar capabilities for writing and scanning strings to and from memory variables. For example, the statement

```
sprintf(dis_strn,"%d %d", num1, num2);
```

writes the numerical values of `num1` and `num2` into `dis_strn` rather than displaying the values on the standard output terminal. Here, `dis_strn` is a programmer-selected variable name that must be declared as either an array of characters sufficiently large to hold the resulting string, or as a pointer to a string.

Typically, the `sprintf( )` function is used to "assemble" a string from smaller pieces until a complete line of characters is ready to be written, either to the standard output device or to a file. For example, another string could be concatenated to `dis_strn` using the `strcat( )` function and the complete string displayed using the `printf( )` function.

In contrast to `sprintf( )`, the string scan function `sscanf( )` may be used to "disassemble" a string into smaller pieces. For example, if the string `"$23.45 10"` were stored in a character array named `data`, the statement

```
sscanf(data,"%c%lf %d",&dol,&price,&units);
```

would scan the string stored in the `data` array and "strip off" three data items. The dollar sign would be stored in the variable named `dol`, the 23.45 would be converted to a double precision number and stored in the variable named `price`, and the 10 would be converted to an integer value and stored in the variable named `units`. For a useful result, the variables `dol`, `price`, and `units` would have to be declared as the appropriate data types. In this way `sscanf( )` provides a useful means of converting parts of a string into other data types. Typically, the string being scanned by `sscanf( )` is used as a working storage area, or buffer, for storing a complete line from either a file or the standard input. Once the string has been filled, `sscanf( )` disassembles the string into component parts and suitably converts each data item into the designated data type. For programmers familiar with COBOL, this is equivalent to first reading data into a working storage area before moving the data into smaller fields.

Format Strings

When you use any of the four functions, `printf( )`, `scanf( )`, `sprintf( )`, or `sscanf( )`, the control string containing the conversion control sequences need not be explicitly contained within the function. For example, the control string `"$%5.2d %d"` contained within the function call

```
printf("$%5.2d %d",num1,num2);
```

can itself be stored as a string and the address of the string used in the call to `printf( )`. If either of the following declarations for `fmat` are made:

```
char *fmat = "$%5.2d %d";
```

or

```
char fmat[ ] = "$%5.2d %d";
```

the function call `printf(fmat,num1,num2);` can be made in place of the previous call to `printf( )`. Here, `fmat` is a pointer that contains the address of the control string used to determine the output display.

The technique of storing and using control strings in this manner is very useful for clearly listing format strings with other variable declarations at the beginning of a function. If a change to a format must be made, it is easy to find the desired control string without the necessity of searching through the complete function to locate the appropriate `printf( )` or `scanf( )` function call. Restricting the definition of a control string to one place is also advantageous when the same format control is used in multiple function calls.

Exercises 11.4

1. Determine the display produced by each of the following statements:
   a. `printf("!%10s!","four score and ten");`
   b. `printf("!%15s!","Home!");`
   c. `printf("!%-15s!,"Home!");`

d. `printf("!%15.2s!","Home!");`
e. `printf("!%-15.2s!","Home!");`

2 *a.* Assuming that the following declaration has been made,

```
char *text = "Have a nice day!";
```

determine the display produced by the statements

```
printf("%s", text);

printf("%c", *text);
```

b. Since both `printf( )` function calls in Exercise 2a display characters, determine why the indirection operator is required in the second call but not in the first.

3. Write a program that accepts three user-entered floating point numbers as one string. Once the string has been accepted, have the program pass the string and the addresses of three floating point variables to a function called `separate( )`. The `separate( )` function should extract the three floating point values from the passed string and store them using the passed variable addresses.

4. Modify the program written for Exercise 3 to display the input string using the format `"%6.2f %6.2f %6.2f"`.

5. Write a program that accepts a string and two integer numbers from a user. Each of these inputs should be preceded by a prompt and stored using individual variable names. Have your program call a function that assembles the input data into a single string. Display the assembled string using a `puts( )` call.

11.5 Common Programming Errors

Four errors are frequently made when pointers to strings are used. The most common is using the pointer to "point to" a nonexistent data element. This error is, of course, the same error we have already seen using subscripts. Since C compilers do not perform bounds checking on arrays, it is the programmer's responsibility to ensure that the address in the pointer is the address of a valid data element.

The second common error occurs when an attempt is made to initialize a local array of characters when using a non-ANSI C compiler. Only `static` local arrays can be initialized in non-ANSI C compilers. Under the ANSI C standard, the local arrays do not have to be `static` to be initialized within their declaration statements. (Both ANSI and non-ANSI C compilers permit character pointers to be initialized without being declared as `static` variables.)

The third common error lies in not providing sufficient space for the end-of-string null character when a string is defined as an array of characters, and not including the `\0` character when the array is initialized.

Finally, the last error relates to a misunderstanding of terminology. For example, if `text` is defined as

```
char *text;
```

the variable `text` is sometimes referred to as a string. Thus, the terminology "store the characters `Hooray for the Hoosiers` into the `text` string" may be encountered. Strictly speaking, calling `text` a string or a string variable is incorrect. The variable `text` is a pointer that contains the address of the first character in the string. Nevertheless, referring to a character pointer as a string occurs frequently enough that you should be aware of it.

11.6 Chapter Summary

1. A string is an array of characters that is terminated by the null character.
2. Strings can always be processed using standard array-processing techniques. The input and display of a string, however, always require reliance on a standard library function.
3. The `gets( )`, `scanf( )`, and `getchar( )` library routines can be used to input a string. The `scanf( )` function tends to be of limited usefulness for string input because it terminates input when encountering a blank.
4. The `puts( )`, `printf( )`, and `putchar( )` routines can be used to display strings.
5. In place of subscripts, pointer notation and pointer arithmetic are especially useful for manipulating string elements.
6. Many standard library functions exist for processing strings as a complete unit. Internally, these functions manipulate strings in a character-by-character manner, usually using pointers.
7. String storage can be created by declaring an array of characters or a pointer to a character. A pointer to a character can be assigned a string directly. String assignment to a string declared as an array of characters is invalid except within a declaration statement.
8. Arrays can be initialized using a string assignment of the form

```
char *arr_name[ ] = "text";
```

This initialization is equivalent to:

```
char *arr_name[ ] = {'t','e','x','t','\0'};
```

Structures

Chapter Twelve

12.1 Single Structures
12.2 Arrays of Structures
12.3 Passing and Returning Structures
12.4 Linked Lists
12.5 Dynamic Storage Allocation
12.6 Unions
12.7 Common Programming Errors
12.8 Chapter Summary

Name:
Street Address:
City:
State:
Zip Code:

FIGURE 12–1 Typical Mailing List Components

In the broadest sense, structure refers to the way individual elements of a group are arranged or organized. For example, a corporation's structure refers to the organization of the people and departments in the company and a government's structure refers to its form or arrangement. In C, a structure refers to the way individual data items are arranged to form a cohesive and related unit. For example, consider the data items typically used in preparing mailing labels, as illustrated in Figure 12–1.

Each of the individual data items listed in the figure is an entity by itself. Taken together, all the data items form a single unit, representing a natural organization of the data for a mailing label. This larger grouping of related individual items is commonly called a record. In C, a record is referred to as a *structure.*

Although there could be thousands of names and addresses in a complete mailing list, the form of each mailing label, or its structure, is identical. In dealing with structures it is important to distinguish between the form of the structure and the data content of the structure.

The form of a structure consists of the symbolic names, data types, and arrangement of individual data items in the structure. The content of a structure refers to the actual data stored in the symbolic names. Figure 12–2 shows acceptable contents for the structure illustrated in Figure 12–1.

In this chapter we describe the C statements required to create, fill, use, and pass structures between functions.

12.1 Single Structures

Using structures requires the same two steps needed for using any C variable. First the structure must be declared. Then specific values can be assigned to the

FIGURE 12–2 The Contents of a Structure

Rhona Bronson-Karp
614 Freeman Street
Orange
NJ
07052

individual structure elements. Declaring a structure requires listing the data types, data names, and arrangement of data items. For example, the definition

```
struct
{
  int month;
  int day;
  int year;
}  birth;
```

gives the form of a structure called `birth` and reserves storage for the individual data items listed in the structure. The `birth` structure consists of three data items, which are called members of the structure.

Assigning actual data values to the data items of a structure is called populating the structure, and is a relatively straightforward procedure. Each member of a structure is accessed by giving both the structure name and individual data item name, separated by a period. Thus, `birth.month` refers to the first member of the birth structure, `birth.day` refers to the second member of the structure, and `birth.year` refers to the third member. Program 12-1 illustrates assigning values to the individual members of the `birth` structure (observe that the `printf()` statement call has been continued across two lines).

Program 12-1

```
#include <stdio.h>
main( )
{
  struct
  {
    int month;
    int day;
    int year;
  } birth;

  birth.month = 12;
  birth.day = 28;
  birth.year = 72;

  printf("My birth date is %d/%d/%d",
         birth.month, birth.day, birth.year);
}
```

The output produced by Program 12-1 is:

```
My birth date is 12/28/72
```

As in most C statements, the spacing of a structure definition is not rigid. For example, the `birth` structure could just as well have been defined as

```
struct {int month; int day; int year} birth;
```

Also, as with all C definition statements, multiple variables can be defined in the same statement. For example, the definition statement

```
struct {int month; int day; int year} birth, current;
```

creates two structures having the same form. The members of the first structure are referenced by the individual names `birth.month`, `birth.day`, and `birth.year`, while the members of the second structure are referenced by the names `current.month`, `current.day`, and `current.year`. Notice that the form of this particular structure definition statement is identical to the form used in defining any program variable. The data type is followed by a list of variable names.

A useful modification of defining structures is listing the form of the structure with no following variable names. In this case, however, the list of structure members must be preceded by a tag name. For example, in the declaration

```
struct date
{
  int month;
  int day;
  int year;
};
```

the term `date` is a tag name. The declaration for the `date` structure provides a template for the structure without actually reserving any storage locations. As such it is not a definition statement. The template presents the form of a structure called `date` by describing how individual data items are arranged within the structure. Actual storage for the members of the structure is reserved only when specific variable names are assigned. For example, the definition statement

```
struct date birth, current;
```

reserves storage for two structures named `birth` and `current`, respectively. Each of these individual structures has the form previously declared for the `date` structure. In effect, the declaration for `date` creates a structure type named `date`. The variables `birth` and `current` are then defined to be of this structure type.

Like all variable declarations, a structure may be declared globally or locally. Program 12-2 illustrates the global declaration of a `date` struc-ture. Internal to `main( )`, the variable `birth` is defined to use the global template.

The output produced by Program 12-2 is identical to the output produced by Program 12-1.

The initialization of structures follows the same rules as for the initialization

Program 12-2

```
#include <stdio.h>
struct date
{
  int month;
  int day;
  int year;
};
main(<)
{
  struct date   birth;

  birth.month = 12;
  birth.day = 28;
  birth.year = 52;

  printf("My birth date is %d/%d/%d",
         birth.month, birth.day, birth.year);
}
```

of arrays: Global and local structures may be initialized by following the definition with a list of initializers[1]. For example, the definition statement

```
struct date birth = {12, 28, 72};
```

can be used to replace the first four statements internal to `main( )` in Program 12-2. Notice that the initializers are separated by commas, not semicolons.

The individual members of a structure are not restricted to integer data types, as illustrated by the `birth` structure. Any valid C data type can be used. For example, consider an employee record consisting of the following data items:

Name:
Identification Number:
Regular Pay Rate:
Overtime Pay Rate:

A suitable declaration for these data items is:

```
struct pay_rec
{
  char name[20];
```

[1] This is true for ANSI C compilers. For non-ANSI C compilers the keyword `static` must be placed before the keyword `struct` for initialization within the declaration statement. This is because `static` local structures may be initialized, whereas automatic local structures cannot be initialized.

```
    int id_num;
    float reg_rate;
    float ot_rate;
};
```

Once the template for `pay_rec` is declared, a specific structure using the `pay_rec` template can be defined and initialized. For example, the definition

```
struct pay_rec employee = {"H. Price",12387,15.89,25.50};
```

creates a structure named `employee` using the `pay_rec` template. The individual members of `employee` are initialized with the respective data listed between braces in the definition statement.

Notice that a single structure is simply a convenient method for combining and storing related items under a common name. Although a single structure is useful in explicitly identifying the relationship among its members, the individual members could be defined as separate variables. The real advantage to using structures is only realized when the same template is used in a list many times over. Creating lists with the same structure template is the topic of the next section.

Before leaving single structures, it is worth noting that the individual members of a structure can be any valid C data type, including both arrays and structures. An array of characters was used as a member of the `employee` structure defined previously. Accessing an element of a member array requires giving the structure's name, followed by a period, followed by the array designation. For example, `employee.name[4]` refers to the fifth character in the `employee` array.

Including a structure within a structure follows the same rules for including any data type in a structure. For example, assume that a structure is to consist of a name and a date of birth, where a `date` structure has been declared as:

```
struct date
{
  int month;
  int date;
  int year;
};
```

A suitable definition of a structure that includes a `name` and a `date` structure is:

```
struct
{
  char name[20];
  struct date birth;
} person;
```

Notice that in declaring the date structure, the term `date` is a structure tag name. A tag name always appears before the braces in the declaration statement and identifies a structure template. In defining the `person` structure, `person` is the name of a specific structure, not a structure tag name. The same is true of the variable named `birth`. This is the name of a specific structure having the form

of `date`. Individual members in the `person` structure are accessed by preceding the desired member with the structure name followed by a period. For example, `person.birth.month` refers to the `month` variable in the `birth` structure contained in the `person` structure.

Exercises 12.1

1. Declare a structure template named `s_temp` for each of the following records:
   a. a student record consisting of a student identification number, number of credits completed, and cumulative grade point average
   b. a student record consisting of a student's name, date of birth, number of credits completed, and cumulative grade point average
   c. a mailing list consisting of the items previously illustrated in Figure 12–1
   d. a stock record consisting of the stock's name, the price of the stock, and the date of purchase
   e. an inventory record consisting of an integer part number, part description, number of parts in inventory, and an integer reorder number

2. For the individual structure templates declared in Exercise 1, define a suitable structure variable name, and initialize each structure with the appropriate following data:
   a. Identification Number: 4672
   Number of Credits Completed: 68
   Grade Point Average: 3.01
   b. Name: Rhona Karp
   Date of Birth: 8/4/60
   Number of Credits Completed: 96
   Grade Point Average: 3.89
   c. Name: Kay Kingsley
   Street Address: 614 Freeman Street
   City: Indianapolis
   State: IN
   Zip Code: 07030
   d. Stock: IBM
   Price Purchased: 134.5
   Date Purchased: 10/1/86
   e. Part Number: 16879
   Description: Battery
   Number in Stock: 10
   Reorder Number: 3

3 *a.* Write a C program that prompts a user to input the current month, day, and year. Store the data entered in a suitably defined structure and display the date in an appropriate manner.
   b. Modify the program written in Exercise 3a to accept the current time in hours, minutes, and seconds.

4. Write a C program that uses a structure for storing the name of a stock, its estimated earnings per share, and its estimated price-to-earnings ratio. Have the program prompt the user to enter these items for five different stocks, each time using the same structure to store the entered data. When the data have been entered for a particular stock, have the program compute and display the anticipated stock price based on the entered earnings and price-per-earnings values. For example, if a user entered the data `XYZ 1.56 12`, the anticipated price for a share of XYZ stock is (1.56) · (12) = $18.72.

5. Write a C program that accepts a user-entered time in hours and minutes. Have the program store the data in a structure, calculate and display the time one minute later.

6 *a.* Write a C program that accepts a user-entered date. Have the program store the data in a structure, calculate and display the date of the next day. For purposes of this exercise, assume that all months consist of 30 days.

b. Modify the program written in Exercise 6a to account for the actual number of days in each month.

12.2 Arrays of Structures

The real power of structures is realized when the same structure is used for lists of data. For example, assume that the data shown in Figure 12–3 must be processed.

Clearly, the employee numbers can be stored together in an array of integers, the names in an array of character arrays, and the pay rates in an array of either floating point or double precision numbers. In organizing the data in this fashion, each column in Figure 12–3 is considered as a separate list, which is stored in its own array. The correspondence between items for each individual employee is maintained by storing an employee's data in the same array position in each array.

The separation of the complete list into three individual arrays is unfortunate,

FIGURE 12–3 A List of Employee Data

Employee number	Employee name	Employee pay rate
32479	Abrams, B.	6.72
33623	Bohm, P.	7.54
34145	Donaldson, S.	5.56
35987	Ernst, T.	5.43
36203	Gwodz, K.	8.72
36417	Hanson, H.	7.64
37634	Monroe, G.	5.29
38321	Price, S.	9.67
39435	Robbins, L.	8.50
39567	Williams, B.	7.20

	Employee number	Employee name	Employee pay rate
1st record ⟶	32479	Abrams, B.	6.72
2nd record ⟶	33623	Bohm, P.	7.54
3rd record ⟶	34145	Donaldson, S.	5.56
4th record ⟶	35987	Ernst, T.	5.43
5th record ⟶	36203	Gwodz, K.	8.72
6th record ⟶	36417	Hanson, H.	7.64
7th record ⟶	37634	Monroe, G.	5.29
8th record ⟶	38321	Price, S.	9.67
9th record ⟶	39435	Robbins, L.	8.50
10th record ⟶	39567	Williams, B.	7.20

FIGURE 12–4 A List of Records

since all of the items relating to a single employee constitute a natural organization of data into records, as illustrated in Figure 12–4.

Using a structure, the program can maintain and reflect the integrity of the data organization as a record. Under this approach, the list illustrated in Figure 12–4 can be processed as a single array of ten structures.

Declaring an array of structures is the same as declaring an array of any other variable type. For example, if the template `pay_rec` is declared as:

```
struct pay_rec {int idnum; char name[20]; float rate;};
```

then an array of ten such structures can be defined as:

```
struct pay_rec employee[10];
```

This definition statement constructs an array of ten elements, each of which is a structure of the type `pay_rec`. Notice that the creation of an array of ten structures has the same form as the creation of any other array. For example, creating an array of ten integers named `employee` requires the declaration

```
int employee[10];
```

In this declaration the data type is integer, while in the former declaration for `employee` the data type is a structure using the `pay_rec` template.

Once an array of structures is declared, a particular data item is referenced

by giving the position of the desired structure in the array followed by a period and the appropriate structure member. For example, the variable `employee[0].rate` references the `rate` member of the first `employee` structure in the `employee` array. Including structures as elements of an array permits a list of records to be processed using standard array programming techniques. Program 12-3 displays the first five employee records illustrated in Figure 12–4.

Program 12-3

```
#include <stdio.h>
struct pay_rec
{
  long id;
  char name[20];
  float rate;
};          /* construct a global template */
main( )
{
  int i;
  struct pay_rec employee[5] =
         {
           { 32479, "Abrams, B.", 6.72 },
           { 33623, "Bohm, P.", 7.54},
           { 34145, "Donaldson, S.",  5.56},
           { 35987, "Ernst, T.", 5.43 },
           { 36203, "Gwodz, K.", 8.72 }
         };

  for ( i = 0; i < 5; ++i)
    printf("\n%ld %-20s %4.2f",employee[i].id,
          employee[i].name,employee[i].rate);
}
```

The output displayed by Program 12-3 is:

```
32479 Abrams, B.          6.72
33623 Bohm, P.            7.54
34145 Donaldson, S.       5.56
35987 Ernst, T.           5.43
36203 Gwodz, K.           8.72
```

In reviewing Program 12-3, notice the initialization of the array of structures. Although the initializers for each structure have been enclosed in inner braces, these are not strictly necessary. The `%-20s` format included in the `printf( )` function call forces each name to be displayed left justified in a field of 20 spaces.

Exercises 12.2

1. Define arrays of 100 structures for each of the structures described in Exercise 1 of the previous section.

2 a. Using the template

```
struct mon_days
{
  char name[10];
  int days;
};
```

define an array of 12 structures of type `mon_days`. Name the array `convert[]`, and initialize the array with the names of the 12 months in a year and the number of days in each month.

b. Include the array created in Exercise 2a in a program that displays the names and number of days in each month.

3. Using the structure defined in Exercise 2a, write a C program that accepts a month from a user in numerical form and displays the name of the month and the number of days in the month. Thus, in response to an input of three, the program would display `March has 31 days.`

4 a. Declare a single structure template suitable for an employee record of the type illustrated below:

Number	Name	Rate	Hours
3462	Jones	4.62	40
6793	Robbins	5.83	38
6985	Smith	5.22	45
7834	Swain	6.89	40
8867	Timmins	6.43	35
9002	Williams	4.75	42

b. Using the template declared in Exercise 4a, write a C program that interactively accepts the above data into an array of six structures. Once the data have been entered, the program should create a payroll report listing each employee's name, number, and gross pay. Include the total gross pay of all employees at the end of the report.

5 a. Declare a single structure template suitable for a car record of the type illustrated:

Car Number	Miles Driven	Gallons Used
25	1,450	62
36	3,240	136
44	1,792	76
52	2,360	105
68	2,114	67

b. Using the template declared for Exercise 5a, write a C program that interactively accepts the above data into an array of five structures. Once the data have been entered, the program should create a report listing each car number and the miles per gallon achieved by the car. At the end of the report include the average miles per gallon achieved by the complete fleet of cars.

12.3 Passing and Returning Structures

Individual structure members may be passed to a function in the same manner as any scalar variable. For example, given the structure definition

```
struct
{
  int id_num;
  double pay_rate;
  double hours;
} emp;
```

the statement

```
display(emp.id_num);
```

passes a copy of the structure member `emp.id_num` to a function named `display( )`. Similarly, the statement

```
calc_pay(emp.pay_rate,emp.hours);
```

passes copies of the values stored in structure members `emp.pay_rate` and `emp.hours` to the `function calc_pay( )`. Both functions, `display( )` and `calc_pay`, must declare the correct data types of their respective arguments.

Complete copies of all members of a structure can also be passed to a function by including the name of the structure as an argument to the called function.[2] For example, the function call

```
calc_net(emp);
```

passes a copy of the complete `emp` structure to `calc_net( )`. Internal to `calc_net( )`, an appropriate declaration must be made to receive the structure. Program 12-4 declares a global template for an employee record. This template is then used by both the `main( )` and `calc_net( )` functions to define specific structures with the names emp and temp, respectively.

The output produced by Program 12-4 is:

```
The net pay for employee 46782 is $361.66
```

In reviewing Program 12-4, observe that both `main( )` and `calc_net( )` use the same global template to define their individual structures. The structure defined in `main( )` and the structure defined in `calc_net( )` are two completely different structures. Any changes made to the local `temp` structure in `calc_net( )` are not reflected in the `emp` structure of `main( )`. In fact, since both structures are local to their respective functions, the same structure name could have been used in both functions with no ambiguity.

[2] ANSI C permits complete structures to be function arguments. Many non-ANSI C compilers do not support this feature.

Program 12-4

```
#include <stdio.h>
struct  employee        /* declare a global template */
{
  int id_num;
  double pay_rate;
  double hours;
};
main( )
{
  struct employee emp = {6782, 8.93, 40.5};
  double net_pay;
  double calc_net(struct employee);      /* function prototype */

  net_pay = calc_net(emp);        /* pass copies of the values in emp */
  printf("The net pay for employee  %d  is  $%6.2f",emp.id_num,net_pay);
}

double calc_net(struct employee temp) /* temp is of data type struct
                                         employee */
{
  return(temp.pay_rate * temp.hours);
}
```

When `calc_net( )` is called by `main( )`, copies of emp's structure values are passed to the temp structure. `calc_net( )` then uses two of the passed member values to calculate a number, which is returned to `main( )`. Since `calc_net( )` returns a noninteger number, the data type of the value returned must be included in all declarations for `calc_net( )`.

Although the structures in both `main( )` and `calc_net( )` use the same globally defined template, this is not strictly necessary. For example, the structure in `main( )` could have been defined directly as:

```
static struct
{
  int id_num;
  double pay_rate;
  double hours;
}  emp = {46782, 8.93, 40.5};
```

Similarly, the structure in `calc_net( )` could have been defined as:

```
struct
{
  int id_num;
  double pay_rate;
  double hours;
} temp;
```

The global declaration of the employee template provided in Program 12-4 is highly preferable to these latter two individual structure specifications because the global template centralizes the declaration of the structure's organization. Any change that must subsequently be made to the structure need only be made once to the global template. Making changes to individual structure definitions requires that all occurrences of the structure definition be located in every function defining the structure. In larger programs this usually results in an error when a change to one of the structure definitions is inadvertently omitted.

An alternative to passing a copy of a structure is to pass the address of the structure. This, of course, allows the called function to make changes directly to the original structure. For example, referring to Program 12-4, the call to `calc_net( )` can be modified to:

```
calc_net(&emp);
```

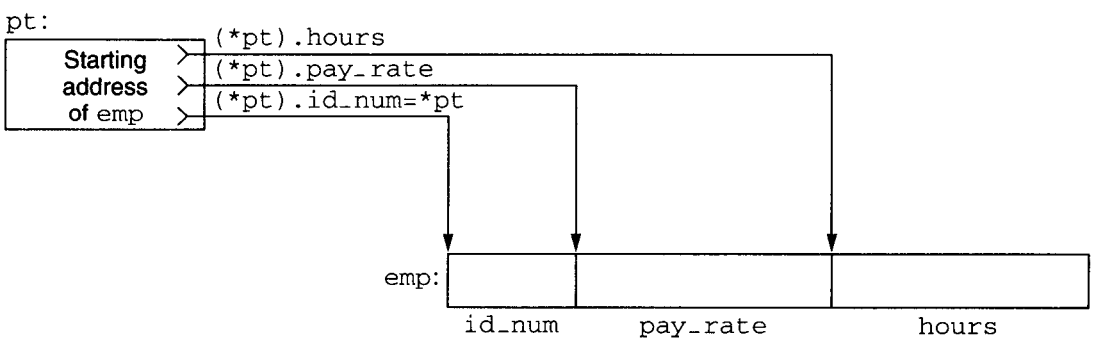
In this call, an address is passed. To correctly store this address `calc_net( )` must declare the argument as a pointer. A suitable function definition for `calc_net( )` is:

```
calc_net(struct employee *pt)
```

Here, the declaration for `pt` declares this argument as a pointer to a structure of type `employee`. The pointer variable, `pt`, receives the starting address of a structure whenever `calc_net( )` is called. Within `calc_net( )`, this pointer is used to directly reference any member in the structure. For example, `(*pt).id_num` refers to the `id_num` member of the structure, `(*pt).pay_rate` refers to the `pay_rate` member of the structure, and `(*pt).hours` refers to the `hours` member of the structure. These relationships are illustrated in Figure 12–5.

The parentheses around the expression `*pt` in Figure 12–5 are necessary to initially access "the structure whose address is in `pt`." This is followed by a reference to access the desired member within the structure. In the absence of the parentheses, the structure member operator . takes precedence over the indirection operator. Thus, the expression `*pt.hours` is another way of writing `*(pt.hours)`, which would refer to "the variable whose address is in the `pt.hours` variable." This last expression clearly makes no sense because there is no structure named `pt` and `hours` does not contain an address.

FIGURE 12–5 A Pointer Can Be Used to Access Structure Members



As illustrated in Figure 12–5, the starting address of the emp structure is also the address of the first member of the structure. Thus, the expressions *pt and (*pt).id_num both refer to the id_num member of the emp structure.

The use of pointers is so common with structures that a special notation exists for them. The general expression (*pointer).member can always be replaced with the notation pointer->member, where the -> operator is constructed using a minus sign followed by a right-facing arrow (greater-than symbol). Either expression can be used to locate the desired member. For example, the following expressions are equivalent:

(*pt).id_num can be replaced by pt->id_num
(*pt).pay_rate can be replaced by pt->pay_rate
(*pt).hours can be replaced by pt->hours

Program 12-5 illustrates passing a structure's address and using a pointer with the new notation to directly reference the structure.

Program 12-5

```
#include <stdio.h>
struct  employee        /* declare a global template */
{
  int id_num;
  double pay_rate;
  double hours;
};
main()
{
  static struct employee emp = {6782, 8.93, 40.5};
  double net_pay;
  double calc_net(struct employee *);     /* function prototype */

  net_pay = calc_net(&emp);         /* pass copies of the values in emp */
  printf("The net pay for employee  %d  is  $%6.2f",emp.id_num,net_pay);
}

double calc_net(struct employee *pt) /* pt is a pointer to a structure */
                                     /* of employee type */
{
  return(pt->pay_rate * pt->hours);
}
```

The name of the pointer argument declared in Program 12-5 is, of course, selected by the programmer. When calc_net( ) is called, emp's starting address is passed to the function. Using this address as a reference point, individual members of the structure are accessed by including their names with the pointer.

As with all C expressions that reference a variable, the increment and decrement operators can also be applied to structure references. For example, the expression

```
++pt->hours
```

adds one to the `hours` member of the `emp` structure. Since the `->` operator has a higher priority than the increment operator, the `hours` member is accessed first and then the increment is applied.

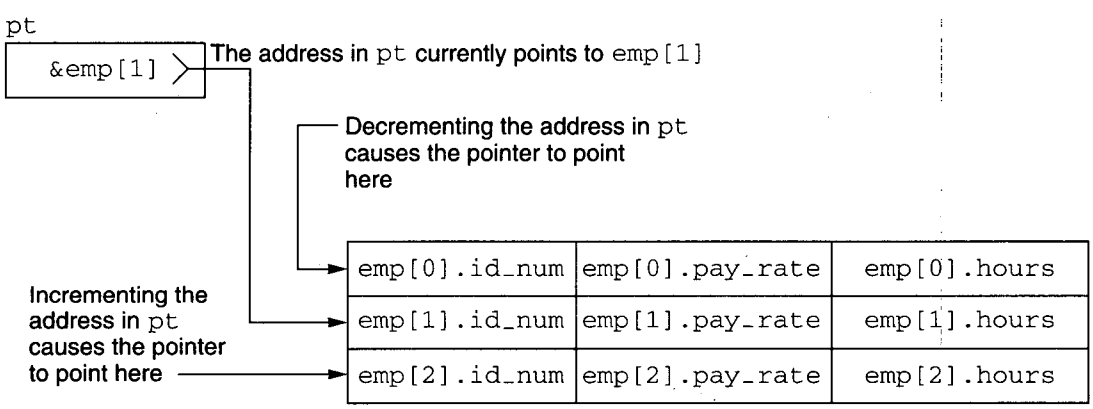
Alternatively, the expression `(++pt)->hours` uses the prefix increment operator to increment the address in `pt` before the `hours` member is accessed. Similarly, the expression `(pt++)->hours` uses the postfix increment operator to increment the address in `pt` after the `hours` member is accessed. In both of these cases, however, there must be sufficient defined structures to ensure that the incremented pointers actually point to legitimate structures.

As an example, Figure 12–6 illustrates an array of three structures of type `employee`. Assuming that the address of `emp[1]` is stored in the pointer variable `pt`, the expression `++pt` changes the address in `pt` to the starting address of `emp[2]`, while the expression `--pt` changes the address to point to `emp[0]`.

Returning Structures

In practice, most structure-handling functions receive direct access to a structure by passing the address of the structure to the function. Then any changes can be made directly by the function using pointer references. If you want to have a function return a separate structure, however, you must follow the same procedures for returning complete structures as for returning scalar values.[3] These procedures include declaring the function appropriately and alerting any calling function to the type of structure being returned. For example, the function `get_vals( )` in Program 12-6 returns a complete structure to `main( )`.

FIGURE 12–6 Changing Pointer Addresses



[3] ANSI C permits structures to be a returned data type. Many non-ANSI C compilers do not permit the return of a structure.

Program 12-6

```
#include <stdio.h>
struct  employee        /* declare a global template */
{
  int id_num;
  double pay_rate;
  double hours;
};
main( )
{
  struct employee emp;
  struct employee get_vals(void);        /* function prototype */

  emp = get_vals( );
  printf("\nThe employee id number is %d", emp.id_num);
  printf("\nThe employee pay rate is $%5.2f", emp.pay_rate);
  printf("\nThe employee hours are %5.2f", emp.hours);
}

struct employee get_vals( ) /*get_vals( ) returns an employee struct*/
{
  struct employee new;

  new.id_num = 6789;
  new.pay_rate = 16.25;
  new.hours = 38.0;
  return(new);
}
```

The following output is displayed when Program 12-6 is run:

```
The employee id number is 56789
The employee pay rate is $16.25
The employee hours are 38.00
```

Since the `get_vals( )` function returns a structure, the function header for `get_vals( )` must specify the type of structure being returned. As `get_vals( )` does not receive any arguments, the function header has no argument declarations and consists of the line

```
struct employee get_vals( )
```

Within `get_vals( )`, the variable `new` is defined as a structure of the type to be returned. After values have been assigned to the `new` structure, the struc-

ture values are returned by including the structure name within the parentheses of the return statement.

On the receiving side, `main( )` must be alerted that the function `get_vals( )` will be returning a structure. This is handled by including a function declaration for `get_vals( )` in `main( )`. Notice that these steps for returning a structure from a function are identical to the normal procedures for returning scalar data types previously described in Chapter 7.

Exercises 12.3

1. Write a C function named `days( )` that determines the number of days from the turn of the century for any date passed as a structure. The date structure should use the template

```
struct date
{
  int month;
  int day;
  int year;
};
```

In writing the `days( )` function, use the convention that all years have 360 days and each month consists of 30 days. The function should return the number of days for any date structure passed to it. Make sure to declare the returned variable a long integer to reserve sufficient room for dates such as 12/19/89.

2. Write a C function named `dif_days( )` that calculates and returns the difference between two dates. Each date is passed to the function as a structure using the following global template:

```
struct date
{
  int month;
  int day;
  int year;
};
```

The `dif_days( )` function should make two calls to the `days( )` function written for Exercise 1.

3. Rewrite the `days( )` function written for Exercise 1 to receive a pointer to a `date` structure, rather than a copy of the complete structure.

4 a. Write a C function named `larger( )` that returns the later date of any two dates passed to it. For example, if the dates 10/9/92 and 11/3/92 are passed to `larger( )`, the second date would be returned.

b. Include the `larger( )` function that was written for Exercise 4a in a complete program. Store the `date` structure returned by `larger( )` in a separate date structure and display the member values of the returned date.

5 a. Modify the function `days( )` written for Exercise 1 to account for the actual number of days in each month. Assume, however, that each year contains 365 days (that is, do not account for leap years).

b. Modify the function written for Exercise 5a to account for leap years.

12.4 Linked Lists

A classic data-handling problem is making additions or deletions to existing records that are maintained in a specific order. This is best illustrated by considering the alphabetical telephone list shown in Figure 12–7. Starting with this initial set of names and telephone numbers, we desire to add new records to the list in the proper alphabetical sequence, and to delete existing records in such a way that the storage for deleted records is eliminated.

Although the insertion or deletion of ordered records can be accomplished using an array of structures, these arrays are not efficient representations for adding or deleting records internal to the array. Arrays are fixed and prespecified in size. Deleting a record from an array creates an empty slot that requires either special marking or shifting up all elements below the deleted record to close the empty slot. Similarly, adding a record to the body of an array of structures requires that all elements below the addition be shifted down to make room for the new entry; or the new element could be added to the bottom of the existing array and the array then resorted to restore the proper order of the records. Thus, either adding or deleting records to such a list generally requires restructuring and rewriting the list — a cumbersome, time-consuming, and inefficient practice.

A linked list provides a convenient method for maintaining a constantly changing list, without the need to continually reorder and restructure the complete list. A linked list is simply a set of structures in which each structure contains at least one member whose value is the address of the next logically ordered structure in the list. Rather than requiring each record to be physically stored in the proper order, each new record is physically added either to the end of the existing list, or wherever the computer has free space in its storage area. The records are "linked" together by including the address of the next record in the record immediately preceding it. From a programming standpoint, the current record being processed contains the address of the next record, no matter where the next record is actually stored.

The concept of a linked list is illustrated in Figure 12–8. Although the actual data for the Lanfrank structure illustrated in the figure may be physically stored anywhere in the computer, the additional member included at the end of the

FIGURE 12–7 A Telephone List in Alphabetical Order

Acme, Sam
(201) 898-2392

Dolan, Edith
(213) 682-3104

Lanfrank, John
(415) 718-4581

Mening, Stephen
(914) 382-7070

Zemann, Harold
(718) 219-9912

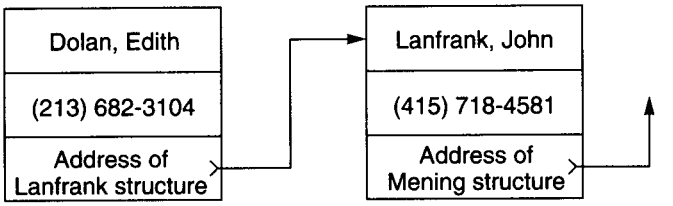


FIGURE 12–8 Using Pointers to Link Structures

Dolan structure maintains the proper alphabetical order. This member provides the starting address of the location where the Lanfrank record is stored. As you might expect, this member is a pointer.

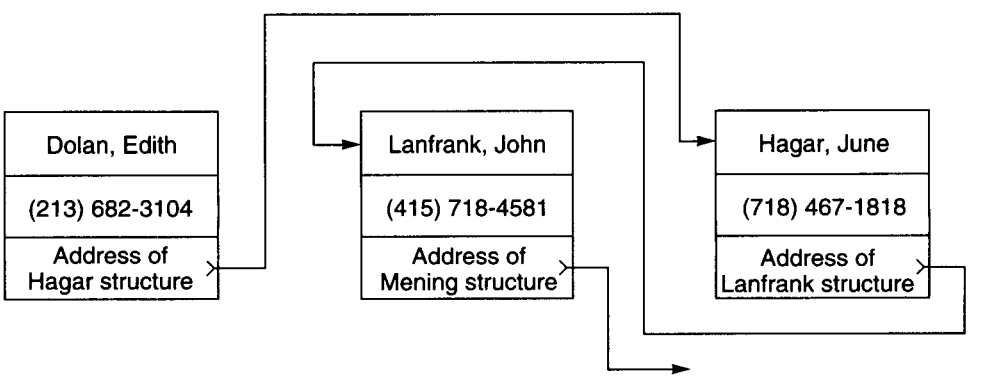
To see the usefulness of the pointer in the Dolan record, let us add a telephone number for June Hagar into the alphabetical list shown in Figure 12–7. The data for June Hagar are stored in a data structure using the same template as that used for the existing records. To ensure that the telephone number for Hagar is correctly displayed after the Dolan telephone number, the address in the Dolan record must be altered to point to the Hagar record, and the address in the Hagar record must be set to point to the Lanfrank record. This is illustrated in Figure 12–9. Notice that the pointer in each structure simply points to the location of the next ordered structure, even if that structure is not physically located in the correct order.

Removal of a structure from the ordered list is the reverse process of adding a record. The actual record is logically removed from the list by simply changing the address in the structure preceding it to point to the structure immediately following the deleted record.

Each structure in a linked list has the same format; however, it is clear that the last record cannot have a valid pointer value that points to another record, since there is none. C provides a special pointer value called `NULL` that acts as a sentinel or flag to indicate when the last record has been processed. The `NULL` pointer value, like its end-of-string counterpart, has a numerical value of zero.

Besides an end-of-list sentinel value, a special pointer must also be provided for storing the address of the first structure in the list. Figure 12–10 illustrates the complete set of pointers and structures for a list consisting of three names.

FIGURE 12–9 Adjusting Addresses to Point to Appropriate Records



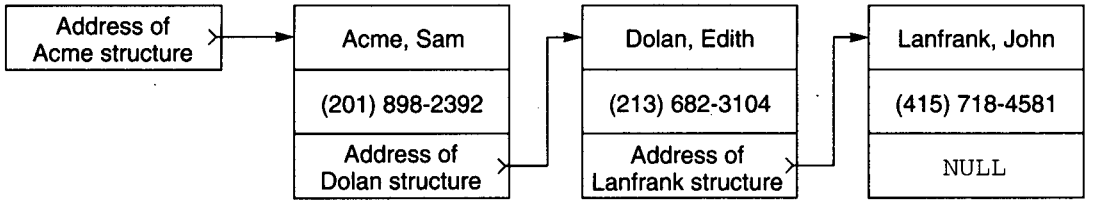


FIGURE 12–10 Use of the Initial and Final Pointer Values

The inclusion of a pointer in a structure should not seem surprising. As we discovered in Section 12.1, a structure can contain any C data type. For example, the structure declaration

```
struct test
{
  int id_num;
  double *pt_pay
};
```

declares a structure template consisting of two members. The first member is an integer variable named `id_num`, and the second variable is a pointer named `pt_pay`, which is a pointer to a double precision number. Program 12-7 illustrates that the pointer member of a structure is used like any other pointer variable.

Program 12-7

```
#include <stdio.h>
struct test
{
  int id_num;
  double *pt_pay;
};
main( )
{
  struct test emp;
  double pay = 456.20;

  emp.id_num = 12345;
  emp.pt_pay = &pay;

  printf("Employee number %d was paid $%6.2f",
       emp.id_num, *emp.pt_pay);
}
```

The output produced by executing Program 12-7 is:

```
Employee number 12345 was paid $456.20
```

Figure 12–11 illustrates the relationship between the members of the emp structure defined in Program 12-7 and the variable named pay. The value assigned to emp.id_num is the number 12345 and the value assigned to pay is 456.20. The address of the pay variable is assigned to the structure member emp.pt_pay. Since this member has been defined as a pointer to a double precision number, placing the address of the double precision variable pay in it is a correct use of this member. Finally, since the member operator. has a higher precedence than the indirection operator *, the expression used in the printf( ) call in Program 12–7 is correct. The expression *emp.pt_pay is equivalent to the expression *(emp.pt_pay), which is translated as "the variable whose address is contained in the member emp.pt_pay."

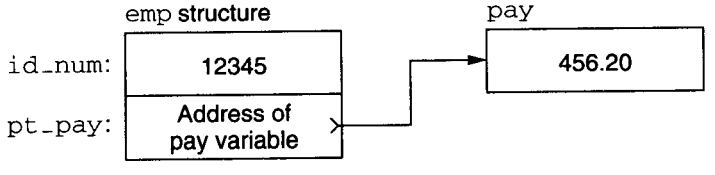
Although the pointer defined in Program 12-7 has been used in a rather trivial fashion, the program does illustrate the concept of including a pointer in a structure. This concept can be easily extended to create a linked list of structures suitable for storing the names and telephone numbers listed in Figure 12–7. The following declaration creates a template for such a structure:

```
struct tele_typ
{
  char name[30];
  char phone_no[15];
  struct tele_typ *nextaddr;
};
```

The tele_typ template consists of three members. The first member is an array of 30 characters, suitable for storing names with a maximum of 29 letters and an end-of-string NULL marker. The next member is an array of 15 characters, suitable for storing telephone numbers with their respective area codes. The last member is a pointer suitable for storing the address of a structure of the tele_typ type.

Program 12-8 illustrates the use of the tele_typ template by specifically defining three structures having this form. The three structures are named t1, t2, and t3, respectively, and the name and telephone members of each of these structures are initialized when the structures are defined, using the data listed in Figure 12–7.

FIGURE 12–11 Storing an Address in a Structure Member



Program 12-8

```
#include <stdio.h>
struct tele_typ
{
  char name[30];
  char phone_no[15];
  struct tele_typ *nextaddr;
};
main( )
{
  struct tele_typ t1 = {"Acme, Sam","(201) 898-2392"};
  struct tele_typ t2 = {"Dolan, Edith","(213) 682-3104"};
  struct tele_typ t3 = {"Lanfrank, John","(415) 718-4581"};
  struct tele_typ *first;    /* create a pointer to a structure */

  first = &t1;              /* store t1's address in first */
  t1.nextaddr = &t2;        /* store t2's address in t1.nextaddr */
  t2.nextaddr = &t3;        /* store t3's address in t2.nextaddr */
  t3.nextaddr = NULL;       /* store the NULL address in t3.nextaddr */

  printf("\n%s \n%s \n%s",first->name,t1.nextaddr->name,t2.nextaddr->name);
}
```

The output produced by executing Program 12-8 is:

```
Acme, Sam
Dolan, Edith
Lanfrank, John
```

Program 12-8 demonstrates the use of pointers to access successive structure members. As illustrated in Figure 12–12, each structure contains the address of the next structure in the list.

The initialization of the names and telephone numbers for each of the structures defined in Program 12-8 is straightforward. Although each structure consists of three members, only the first two members of each structure are initialized. As both of these members are arrays of characters, they can be initialized with strings. The remaining member of each structure is a pointer. To create a linked list, each structure pointer must be assigned the address of the next structure in the list.

The four assignment statements in Program 12-8 perform the correct assignments. The expression `first = &t1` stores the address of the first structure in the list in the pointer variable named `first`. The expression `t1.nextaddr = &t2` stores the starting address of the `t2` structure into the pointer member of the `t1` structure. Similarly, the expression `t2.nextaddr = &t3` stores the starting address of the `t3` structure into the pointer member of the `t2` structure. To end the list, the value of the `NULL` pointer, which is zero, is stored into the pointer member of the `t3` structure.

Once values have been assigned to each structure member and correct

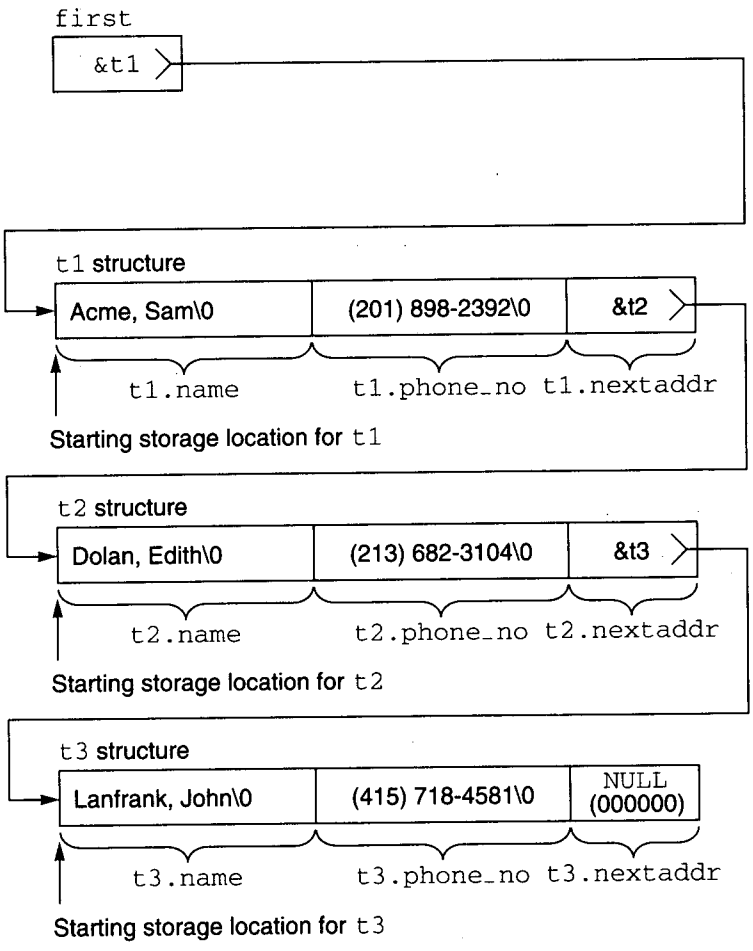


FIGURE 12–12 The Relationship Between Structures in Program 12-8

addresses have been stored in the appropriate pointers, the addresses in the pointers are used to access each structure's name member. For example, the expression `t1.nextaddr->name` refers to the name member of the structure whose address is in the `nextaddr` member of the `t1` structure. The precedence of the member operator `.` and the structure pointer operator `->` are equal, and are evaluated from left to right. Thus, the expression `t1.nextaddr->name` is evaluated as `(t1.nextaddr)->name`. Since `t1.nextaddr` contains the address of the `t2` structure, the proper name is accessed.

The expression `t1.nextaddr->name` can, of course, be replaced by the equivalent expression `(*t1.nextaddr).name`, which uses the more conventional indirection operator. This expression also refers to "the name member of the variable whose address is in `t1.nextaddr`."

The addresses in a linked list of structures can be used to loop through the complete list. As each structure is accessed it can be either examined to select a specific value or used to print out a complete list. For example, the `display( )` function in Program 12-9 illustrates the use of a `while` loop, which uses the address in each structure's pointer member to cycle through the list and successively display data stored in each structure.

Program 12-9

```
#include <stdio.h>
struct tele_typ
{
  char name[30];
  char phone_no[15];
  struct tele_typ *nextaddr;
};
main( )
{
  struct tele_typ t1 = {"Acme, Sam","(201) 898-2392"};
  struct tele_typ t2 = {"Dolan, Edith","(213) 682-3104"};
  struct tele_typ t3 = {"Lanfrank, John","(415) 718-4581"};
  struct tele_typ *first;   /* create a pointer to a structure */
  void display(struct tele_typ *);       /* function prototype */

  first = &t1;          /* store t1's address in first */
  t1.nextaddr = &t2;    /* store t2's address in t1.nextaddr */
  t2.nextaddr = &t3;    /* store t3's address in t2.nextaddr */
  t3.nextaddr = NULL;   /* store the NULL address in t3.nextaddr */

  display(first);       /* send the address of the first structure */
}

void display(struct tele_typ *contents) /* contents is a pointer to a structure */
{                                       /* of type tele_typ                     */
  while (contents != NULL)              /* display till end of linked list */
  {
    printf("\n%-30s %-20s",contents->name, contents->phone_no);
    contents = contents->nextaddr;      /* get next address */
  }
  return;
}
```

The output produced by Program 12-9 is:

```
Acme, Sam                      (201) 898-2392
Dolan, Edith                   (213) 682-3104
Lanfrank, John                 (415) 718-4581
```

The important concept illustrated by Program 12-9 is the use of the address in one structure to access members of the next structure in the list. When the `display( )` function is called, it is passed the value stored in the variable named `first`. Since `first` is a pointer variable, the actual value passed is an address (the address of the `t1` structure). `display( )` accepts the passed value in the argument named `contents`. To store the passed address correctly, `contents` is declared as a pointer to a structure of the `tele_typ` type. Within `display( )`, a `while` loop is used to cycle through the linked structures, starting with the structure whose address is in `contents`. The condition tested in the `while` statement compares the value in `contents`, which is an address, to the `NULL`

value. For each valid address the name and phone number members of the addressed structure are displayed. The address in `contents` is updated with the address in the pointer member of the current structure. The address in `contents` is retested, and the process continues while the address in `contents` is not equal to the NULL value. `display( )` "knows" nothing about the names of the structures declared in `main( )` or even how many structures exist. It simply cycles through the linked list, structure by structure, until it encounters the end-of-list NULL address.

A disadvantage of Program 12-9 is that exactly three structures are defined in `main( )` by name and storage for them is reserved at compile time. Should a fourth structure be required, the additional structure would have to be declared and the program recompiled. In the next section we show how to have the computer dynamically allocate and release storage for structures at run time, as storage is required. Only when a new structure is to be added to the list, and while the program is running, is storage for the new structure created. Similarly, when a structure is no longer needed and can be deleted from the list, the storage for the deleted record is relinquished and returned to the computer.

Exercises 12.4

1. Modify Program 12-9 to prompt the user for a name. Have the program search the existing list for the entered name. If the name is in the list, display the corresponding phone number; otherwise display this message: The name is not in the current phone directory.

2. Write a C program containing a linked list of ten integer numbers. Have the program display the numbers in the list.

3. Using the linked list of structures illustrated in Figure 12–12, write the sequence of steps necessary to delete the record for Edith Dolan from the list.

4. Generalize the description obtained in Exercise 3 to describe the sequence of steps necessary to remove the nth structure from a list of linked structures. The nth structure is preceded by the $(n - 1)$st structure and followed by the $(n + 1)$st structure. Make sure to store all pointer values correctly.

5 *a.* A doubly linked list is a list in which each structure contains a pointer to both the following and previous structures in the list. Define an appropriate template for a doubly linked list of names and telephone numbers.

b. Using the template defined in Exercise 5a, modify Program 12-9 to list the names and phone numbers in reverse order.

12.5 Dynamic Storage Allocation

As each variable is defined in a program, sufficient storage for it is assigned from a pool of computer memory locations made available by the operating system. The size of the assigned storage area is specified by the compiler using the definition statements contained in the program. Once specific memory locations have been reserved for a variable, these locations are fixed for the duration of the pro-

gram, whether they are used or not. For example, if a function requests storage for an array of 500 integers, the storage for the array is allocated and fixed while the program is running. If the application requires less than 500 integers, the unused allocated storage is not released back to the system until the program ends execution. On the other hand, if the application requires more than 500 integers, the size of the integer array must be increased and the function defining the array recompiled.

An alternative to this fixed or static allocation of memory storage locations is the dynamic allocation of memory. Under a dynamic allocation scheme, storage is allocated to a program and released back to the computer while the program is running, rather than being fixed at compile time.

The dynamic allocation of memory is extremely useful when dealing with lists of structures, because it allows the list to expand as new records are added and contract as records are deleted. For example, in constructing a list of names and phone numbers, the exact number of structures ultimately needed may not be known. Rather than creating a fixed array of structures, it is extremely useful to have a mechanism whereby the list can be enlarged and shrunk as necessary. Most standard C libraries provide functions that have this dynamic allocation capability. Two of these functions, called `malloc( )` and `free( )`, are described in Table 12–1.

In requesting additional storage space, the user must provide the `malloc( )` function with an indication of the amount of storage needed. This may be done either by requesting a specific number of bytes or, more usually, by requesting enough space for a particular type of data. For example, the function call `malloc(10)` requests 10 bytes of storage, while the function call `malloc(sizeof(int))` requests enough storage to store an integer number. A request for enough storage for a data structure typically takes the second form. For example, using the structure declaration

```
struct tel_typ
{
  char name[25];
  char phone_no[15];
};
```

the function call `malloc( sizeof(struct tel_typ) );` reserves enough storage for one structure of the `tel_typ` type.

TABLE 12–1

Function Name	Description
`malloc( )`	Reserves the number of bytes requested by the argument passed to the function. Returns the address of the first reserved location or NULL if sufficient memory is not available.
`free( )`	Releases a block of bytes previously reserved. The address of the first reserved location is passed as an argument to the function.

In allocating storage dynamically, we have no advance indication as to where the computer system will physically reserve the requested number of bytes, and we have no explicit name to access the newly created storage locations. To provide access to these locations, `malloc( )` returns the address of the first location that has been reserved. This address must, of course, be assigned to a pointer. The return of a pointer by `malloc( )` is especially useful for creating a linked list of data structures. As each new structure is created, the pointer returned by `malloc( )` to the structure can be assigned to a member of the previous structure in the list. Before illustrating the actual dynamic allocation of such a structure in a program, we will consider one logistic problem created by `malloc( )`.

The `malloc( )` function always returns a pointer to the first byte of storage reserved, and considers this first byte to be a character. Thus, the function declaration of `malloc( )` is `char *malloc( )`. Any function that calls `malloc( )` must include this declaration in order to be alerted that a pointer to a character will be returned. This presents a slight problem when using `malloc( )` to reserve enough storage for a structure. Although `malloc( )` will reserve the necessary number of bytes for a structure and return the correct address of the first reserved byte, this address will be interpreted as the address of a character. To use this address to reference subsequent structure members, it must be reinterpreted as pointing to a structure.

The mechanism for converting one data type into another is the cast operation previously described in section 3.1. In this case, then, we need to cast (or force) a pointer to a character into a pointer to a structure. The cast expression `(struct template_name *)` can be used to do this. For example, if the variable `lis_point` is a pointer to a character, the expression `(struct emp *)lis_point` redefines the address in the pointer to be an address of a structure of type `emp`. The address is not changed physically, but any subsequent reference to the address will now cause the correct number of bytes to be accessed for the appropriate structure. Thus, the cast expressions converts the address returned by `malloc( )` into the correct pointer type for referencing a structure.

Program 12-10 on page 508 illustrates using `malloc( )` to create a structure dynamically in response to a user-input request.

A sample session produced by Program 12-10 is:

```
Do you wish to create a new record (respond with y or n): y
Enter a name: Monroe, James
Enter the phone number: (617) 555-1817

The contents of the record just created is:
Name: Monroe, James
Phone Number: (617) 555-1817
```

In reviewing Program 12-10, notice that only three declarations are made in `main( )`. The variable `key` is declared as a character variable, the `malloc( )` function is declared as providing a pointer to a character, and the variable `rec_point` is declared as being a pointer to a structure of the `tel_typ` type. Since the declaration for the template `tel_typ` is global, `tel_typ` can be used within `main( )` to define `rec_point` as a pointer to a structure of the `tel_typ` type.

Program 12-10

```
#include <stdio.h>
struct tel_typ
{
  name[30];
  phone_no[15];
};
main( )
{
  char key;
  char *malloc( );
  struct tel_typ *rec_point;    /* rec_point is a pointer to a */
                                /* structure of type tel_typ   */
  void populate(struct tel_typ *);  /* funciton protoytpe */
  void disp_one(struct tel_typ *);  /* function prototype */

  printf("Do you wish to create a new record (respond with y or n): ");
  key = getchar( );
  if (key == 'y')
  {
    key = getchar( );       /* get the Enter key in buffered input */
    rec_point = (struct tel_typ *)malloc(sizeof(struct tel_typ));
    populate(rec_point);
    disp_one(rec_point);
  }
  else
    printf("\nNo record has been created.");
}
/* get a name and phone number */
void populate(struct tel_typ *record) /* record is a pointer to a  */
{                                     /* structure of type tel_typ */
printf("Enter a name: ");
gets(record->name);
printf("Enter the phone number: ");
gets(record->phone_no);
return;
}
/* display the contents of one record */
void disp_one(struct tel_typ *contents)/* contents is a pointer to a */
{                                     /* structure of type tel_typ  */
printf("\nThe contents of the record just created is:");
printf("\nName: %s",contents->name);
printf("\nPhone Number: %s", contents->phone_no);
return;
}
```

If a user enters `y` in response to the first prompt in `main( )`, a call to `malloc( )` is made. The argument passed to `malloc( )` is the size of the required structure. Although `malloc( )` returns the address of the first reserved location, this address is considered as pointing to a character. To store the address in

rec_point, which has been declared as a pointer to a structure, the address returned by malloc( ) is coerced into the proper type by use of the expression (struct tel_typ *).

Once rec_point has been loaded with the proper address, this address can be used to access the newly created structure. The function populate( ) is used to prompt the user for data needed in filling the structure and to store the user-entered data in the correct members of the structure. The argument passed to populate( ) in main( ) is the pointer rec_point. Like all passed arguments, the value contained in rec_point is passed to the function. Since the value in rec_point is an address, populate( ) actually receives the address of the newly created structure.

Within populate( ), the value received by it is stored in the argument named record. Since the value to be stored in record is the address of a structure, record must be declared as a pointer to a structure. This declaration is provided by the statement struct tel_typ *record;. The statements within populate( ) use the address in record to locate the respective members of the structure.

The disp_one( ) function in Program 12-10 is used to display the contents of the newly created and populated structure. The address passed to disp_one( ) is the same address that was passed to populate( ). Since this passed value is the address of a structure, the argument name used to store the address is declared as a pointer to the correct structure type.

Once you understand the mechanism of calling malloc( ), you can use this function to construct a linked list of structures. As described in the previous section, the structures used in a linked list must contain one pointer member. The address in the pointer member is the starting address of the next structure in the list. Additionally, a pointer must be reserved for the address of the first structure, and the pointer member of the last structure in the list is given a NULL address to indicate that no more members are being pointed to. Program 12-11 illustrates the use of malloc( ) to construct a linked list of names and phone numbers. The populate( ) function used in Program 12-11 is the same function used in Program 12-10, while the display( ) function is the same function used in Program 12-9.

The first time malloc( ) is called in Program 12-11 it is used to create the first structure in the linked list. As such, the address returned by malloc( ) is stored in the pointer variable named list. The address in list is then assigned to the pointer named current. This pointer variable is always used by the program to point to the current structure. Since the current structure is the first structure created, the address in the pointer named list is assigned to the pointer named current.

Within main( )'s for loop, the name and phone number members of the newly created structure are populated by calling populate( ) and passing the address of the current structure to the function. Upon return from populate( ), the pointer member of the current structure is assigned an address. This address is the address of the next structure in the list, which is obtained from malloc( ). The call to malloc( ) creates the next structure and returns its address into the pointer member of the current structure. This completes the population of the current member. The final statement in the for loop resets the address in the current pointer to the address of the next structure in the list.

After the last structure has been created, the final statements in main( )

Program 12-11

```
#include <stdio.h>
struct tel_typ
{
  char name[25];
  char phone_no[15];
  struct tel_typ *nextaddr;
};
main( )
{
  int i;
  struct tel_typ *list, *current; /* two pointers to structures of  */
                                  /* type tel_typ                   */
  char *malloc( );       /* malloc( ) returns a pointer to a char   */
  void populate(struct tel_typ *);/* function prototype             */
  void display(struct tel_typ *); /* function prototype             */

  /* get a pointer to the first structure in the list */
  list = (struct tel_typ *) malloc(sizeof(struct tel_typ));
  current = list;

  /* populate the current structure and create two more structures */
  for(i = 0; i < 2; ++i)
  {
    populate(current);
    current->nextaddr = (struct  tel_typ *) malloc(sizeof(struct tel_typ));
    current = current->nextaddr;
  }

  populate(current);          /* populate the last structure */
  current->nextaddr = NULL;  /* set the last address */
  printf("\nThe list consists of the following records:\n");
  display(list);              /* display the structures */
}

/* get a name and phone number */
void populate(struct tel_typ *record) /* record is a pointer to a  */
{                                     /* structure of type tel_typ */
  printf("Enter a name: ");
  gets(record->name);
  printf("Enter the phone number: ");
  gets(record->phone_no);
  return;
}

void display(struct tel_typ *contents) /* contents is a pointer to a  */
{                                      /* structure of type tel_typ   */
  while (contents != NULL)             /* display till end of linked list */
  {
    printf("\n%-30s %-20s",contents->name, contents->phone_no);
    contents = contents->nextaddr;
  }
  return;
}
```

populate this structure, assign a NULL address to the pointer member, and call `display( )` to display all the structures in the list. A sample run of Program 12-11 is provided below:

```
Enter a name: Acme, Sam
Enter the phone number: (201) 898-2392
Enter a name: Dolan, Edith
Enter the phone number: (213) 682-3104
Enter a name: Lanfrank, John
Enter the phone number: (415) 718-4581

The list consists of the following records:

Acme, Sam                    (201) 898-2392
Dolan, Edith                 (213) 682-3104
Lanfrank, John               (415) 718-4581
```

Just as `malloc( )` dynamically creates storage while a program is executing, the `free( )` function restores a block of storage back to the computer while the programming is executing. The only argument required by `free( )` is the starting address of a block of storage that was dynamically allocated. Thus, any address returned by `malloc( )` can subsequently be passed to `free( )` to restore the reserved memory back to the computer. `free( )` does not alter the address passed to it, but simply removes the storage that the address references.

Exercises 12.5

1. As described in Table 12–1, the `malloc( )` function returns either the address of the first new storage area allocated, or NULL if insufficient storage is available. Modify Program 12-11 to check that a valid address has been returned before a call to `populate( )` is made. Display an appropriate message if sufficient storage is not available.

2. Write a C function named `delete( )` that deletes an existing structure from the linked list of structures created by Program 12-11. The algorithm for deleting a linked structure should follow the sequence develqped for deleting a structure developed in Exercise 4 in Section 12.4. The argument passed to `delete( )` should be the address of the structure preceding the record to be deleted. In the deletion function, make sure that the value of the pointer in the deleted structure replaces the value of the pointer member of the preceding structure before the structure is deleted.

3. Write a function named `insert( )` that inserts a structure into the linked list of structures created in Program 12-11. The algorithm for inserting a structure in a linked list should follow the sequence for inserting a record illustrated in Figure 12–9. The argument passed to `insert( )` should be the address of the structure preceding the structure to be inserted. The inserted structure should follow this current structure. The `insert( )` function should create a new structure dynamically, call the `populate` function used in Program 12-11, and adjust all pointer values appropriately.

4. We desire to insert a new structure into the linked list of structures created by Program 12-11. The function developed to do this in Exercise 3 assumed that the address of the preceding structure is known. Write a function called `find_rec( )` that returns the address of the structure immediately preceding the point at which the new structure is to be inserted. (*Hint:* `find_rec( )` must request the new name as input and compare the entered name to existing names to determine where to place the new name.)

5. Write a C function named `modify( )` that can be used to modify the name and phone number members of a structure of the type created in Program 12-11. The argument passed to `modify( )` should be the address of the structure to be modified. The `modify( )` function should first display the existing name and phone number in the selected structure and then request new data for these members.

6 *a.* Write a C program that initially presents a menu of choices for the user. The menu should consist of the following choices:

A. Create an initial linked list of names and phone numbers.
B. Insert a new structure into the linked list.
C. Modify an existing structure in the linked list.
D. Delete an existing structure from the list.
E. Exit from the program.

Upon the user's selection, the program should execute the appropriate functions to satisfy the request.

b. Why is the original creation of a linked list usually done by one program, and the options to add, modify, or delete a structure in the list provided by a different program?

12.6 Unions

A union is a data type that reserves the same area in memory for two or more variables, each of which can be a different data type. A variable that is declared as a union data type can be used to hold a character variable, an integer variable, a double precision variable, or any other valid C data type. Each of these types, but only one at a time, can actually be assigned to the union variable.

The declaration for a `union` is identical in form to a structure declaration, with the reserved word union used in place of the reserved word `structure`. For example, the declaration

```
union
{
  char key;
  int num;
  double price;
} val;
```

creates a union variable named `val`. If `val` were a structure it would consist of three individual members. As a union, however, `val` contains a single member that can be either a character variable named `key`, an integer variable named `num`, or a double precision variable named `price`. In effect, a union reserves sufficient memory locations to accommodate its largest member's data type. This same set of locations is then referenced by different variable names depending on the data type of the value currently residing in the reserved locations. Each value stored overwrites the previous value, using as many bytes of the reserved memory area as necessary.

Individual union members are referenced using the same notation as structure members. For example, if the `val` union is currently being used to store a character, the correct variable name to access the stored character is `val.key`. Similarly, if the union is used to store an integer, the value is accessed by the name `val.num`, and a double precision value is accessed by the name `val.price`. In using union members, it is the programmer's responsibility to

ensure that the correct member name is used for the data type currently residing in the union.

Typically a second variable is used to keep track of the current data type stored in the union. For example, the following code could be used to select the appropriate member of `val` for display. Here the value in the variable `u_type` determines the currently stored data type in the `val` union.

```
switch(u_type)
{
  case 'c': printf("%c", val.key);
            break;
  case 'i': printf("%d", val.num);
            break;
  case 'd': printf("%f", val.price);
            break;
  default : printf("Invalid type in u_type : %c", u_type);
}
```

As they are in structures, tag names can be associated with a union to create templates. For example, the declaration

```
union date_time
{
  long int date;
  double time;
};
```

provides a template for a union without actually reserving any storage locations. The template can then be used to define any number of variables of the union type `date_time`. For example, the definition

```
union date_time first, second, *pt;
```

creates a union variable named `first`, a union variable named `second`, and a pointer that can be used to store the address of any union having the form of `date_time`. Once a pointer to a union has been declared, the same notation used to access structure members can be used to access union members. For example, if the assignment `pt = &first;` is made, then `pt->date` references the `date` member of the union named `first`.

Unions may themselves be members of structures or arrays, or structures, arrays, and pointers may be members of unions. In each case, the notation used to access a member must be consistent with the nesting employed. For example, in the structure defined by:

```
struct
{
  int u_type;
  union
  {
    char *text;
    float rate;
  } u_tax
}  flag;
```

the variable `rate` is referenced as:

```
flag.u_tax.rate
```

Similarly, the first character of the string whose address is stored in the pointer `text` is referenced as:

```
*flag.u_tax.text
```

Exercises 12.6

1. Assume that the following definition has been made

```
union
{
  float rate;
  double taxes;
  int num;
} flag;
```

For this union write appropriate `printf( )` function calls to display the various members of the union.

2. Define a union variable named `car` that contains an integer named `year`, an array of 10 characters named `name`, and an array of 10 characters named `model`.

3. Define a union variable named `lang` that would allow a floating point number to be referenced by both the variable names `interest` and `rate`.

4. Declare a union with the tag name `amt` that contains an integer variable named `int_amt`, a double precision variable named `dbl_amt`, and a pointer to a character named `pt_key`.

5 a. What do you think will be displayed by the following section of code?

```
union
{
  char ch;
  float btype;
} alt;
alt.ch = 'y';
printf("%f", alt.btype);
```

b. Include the code presented in Exercise 5a in a program and run the program to verify your answer to Exercise 5a.

12.7 Common Programming Errors

Three common errors are often made when using structures or unions. The first error occurs because structures and unions, as complete entities, cannot be used in relational expressions. For example, even if `tel_typ` and `phon_type` are two

structures of the same type, the expression `tel_typ == phon_typ` is invalid. Individual members of a structure or union can, of course, be compared using any of C's relational operators.

The second common error is really an extension of a pointer error as it relates to structures and unions. Whenever a pointer is used to "point to" either of these data types, or whenever a pointer is itself a member of a structure or a union, take care to use the address in the pointer to access the appropriate data type. Should you be confused about just what is being pointed to, remember, "If in doubt, print it out."

The final error relates specifically to unions. Since a union can store only one of its members at a time, you must be careful to keep track of the currently stored variable. Storing one data type in a union and accessing by the wrong variable name can result in an error that is particularly troublesome to locate.

12.8 Chapter Summary

1. A structure allows individual variables to be grouped under a common variable name. Each variable in a structure is referenced by its structure name, followed by a period, followed by its individual variable name. Another term for a structure is a record. The general form for declaring a structure is:

```
struct
{
   individual member declarations;
} structure_name;
```

2. A tag name can be used to create a generalized structure template describing the form and arrangement of elements in a structure. This tag name can then be used to define specific structure variables.
3. Structures are particularly useful as elements of arrays. Used in this manner, each structure becomes one record in a list of records.
4. Individual members of a structure are passed to a function in the manner appropriate to the data type of the member being passed. ANSI compilers also allow complete structures to be passed, in which case the called function receives a copy of each element in the structure. The address of a structure may also be passed, which provides the called function with direct access to the structure.
5. Structure members can be any valid C data type, including structures, unions, arrays, and pointers. When a pointer is included as a structure member a linked list can be created. Such a list uses the pointer in one structure to "point to" (contain the address of) the next logical structure in the list.
6. Unions are declared in the same manner as structures. The definition of a union creates a memory overlay area, with each union member using the same memory storage locations. Thus, only one member of a union may be active at a time.

Bit Operations

Chapter Thirteen

13.1 The AND Operator
13.2 The Inclusive OR Operator
13.3 The Exclusive OR Operator
13.4 The Complement Operator
13.5 Different-Size Data Items
13.6 The Shift Operators
13.7 Chapter Summary

C operates with complete data entities that are stored as one or more bytes, such as character, integer, and double precision constants and variables. In addition, C provides for the manipulation of individual bits of character and integer constants and variables. Generally these bit manipulations are used in engineering and computer science programs and are not required in commercial applications.[1]

The operators that are used to perform bit manipulations are called bit operators. They are listed in Table 13–1.

All the operators listed in Table 13–1, except ~, are binary operators, requiring two operands. Each operand is treated as a binary number consisting of a series of individual 1s and 0s. The respective bits in each operand are compared on a bit-by-bit basis and the result is determined based on the selected operation.

TABLE 13–1 Bit Operators

Operator	Description
&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise exclusive OR
~	Bitwise one's complement
<<	Left shift
>>	Right shift

13.1 The AND Operator

The AND operator causes a bit-by-bit AND comparison between its two operands. The result of each bit-by-bit comparison is a 1 only when both bits being compared are 1s, otherwise the result of the AND operation is a 0. For example, assume that the following two eight-bit numbers are to be ANDed:

```
1 0 1 1 0 0 1 1
1 1 0 1 0 1 0 1
```

To perform an AND operation, each bit in one operand is compared to the bit occupying the same position in the other operand. Figure 13–1 illustrates the correspondence between bits for these two operands. As shown in the figure, when both bits being compared are 1s, the result is a 1, otherwise the result is a 0. The result of each comparison is, of course, independent of any other bit comparison.

FIGURE 13–1 A Sample AND Operation

```
  1 0 1 1 0 0 1 1
& 1 1 0 1 0 1 0 1
-----------------
  1 0 0 1 0 0 0 1
```

[1] This chapter may be omitted with no loss of subject continuity.

Program 13-1 illustrates the use of an AND operation. In this program, the variable op1 is initialized to the octal value 325, which is the octal equivalent of the binary number 1 1 0 1 0 1 0 1, and the variable op2 is initialized to the octal value 263, which is the octal representation of the binary number 1 0 1 1 0 0 1 1. These are the same two binary numbers illustrated in Figure 13–1.

Program 13-1

```
#include <stdio.h>
main( )
{
  int op1 = 0325, op2 = 0263;

  printf("%o ANDed with %o is %o", op1, op2, op1 & op2);
}
```

Program 13-1 produces the following output:

```
325 ANDed with 263 is 221
```

The result of ANDing the octal numbers 325 and 263 is the octal number 221. The binary equivalent of 221 is the binary number 1 0 0 1 0 0 0 1, which is the result of the AND operation illustrated in Figure 13–1.

AND operations are extremely useful in masking, or eliminating, selected bits from an operand. This is a direct result of the fact that ANDing any bit (1 or 0) with a 0 forces the resulting bit to be a 0, while ANDing any bit (1 or 0) with a 1 leaves the original bit unchanged. For example, assume that the variable op1 has the arbitrary bit pattern x x x x x x x x, where each x can be either 1 or 0, independent of any other x in the number. The result of ANDing this binary number with the binary number 0 0 0 0 1 1 1 1 is:

```
op1 =    x x x x x x x x
op2 =    0 0 0 0 1 1 1 1
         ---------------
Result = 0 0 0 0 x x x x
```

As can be seen from this example, the zeros in op2 effectively mask, or eliminate, the respective bits in op1, while the ones in op2 filter, or pass, the respective bits in op1 through with no change in their values. In this example, the variable op2 is called a mask. By choosing the mask appropriately, any individual bit in an operand can be selected, or filtered, out of an operand for inspection. For example, ANDing the variable op1 with the mask 0 0 0 0 0 1 0 0 forces all the bits of the result to be zero, except for the third bit. The third bit of the result will be a copy of the third bit of op1. Thus, if the result of the AND is zero, the third bit of op1 must have been zero, and if the result of the AND is a nonzero number, the third bit must have been a 1.

Program 13-2 uses this masking property to convert lowercase letters in a word into their its uppercase form, assuming the letters are stored using the ASCII code. The algorithm for converting letters is based on the fact that the binary codes for lowercase and uppercase letters in ASCII are the same except for bit 6, which is 1 for lowercase letters and 0 for uppercase letters. For example, the binary code for the letter a is `01100001` (hex 61), while the binary code for the letter A is `01000001` (hex 41). Similarly, the binary code for the letter z is `01111010` (hex 7A), while the binary code for the letter Z is `01011010` (hex 5A). (See Appendix F for the hexadecimal values of the uppercase and lowercase letters.) Thus, a lowercase letter can be converted into its uppercase form by forcing the sixth bit to zero. This is accomplished in Program 13-2 by masking the letter's code with the binary value `11011111`, which has the hexadecimal value DF.

Program 13-2

```
#include <stdio.h>
main( )
{
  char word[81];       /* enough storage for a complete line */
  void upper(char *); /* function prototype */

  printf("Enter a string of both upper and lowercase letters:\n");
  gets(word);
  printf("\nThe string of letters just entered is:\n");
  puts(word);
  upper(word);
  printf("\nThis string in uppercase letters is:\n");
  puts(word);
}
void upper(char *word)
{
  while (*word != '\0')
    *word++ &= 0XDF;
}
```

A sample run using Program 13-2 follows:

```
Enter a string of both upper and lowercase letters:
abcdefgHIJKLMNOPqrstuvwxyz

The string of letters just entered is:
abcdefgHIJKLMNOPqrstuvwxyz

This string in uppercase letters is:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Notice that the lowercase letters are converted to uppercase form, while uppercase letters are unaltered. This is because bit 6 of all uppercase letters is zero to begin with, so that forcing this bit to zero using the mask has no effect. Only when bit 6 is a one, as it is for lowercase letters, is the input character altered.

13.2 The Inclusive OR Operator

The inclusive OR operator, |, performs a bit-by-bit comparison of its two operands in a fashion similar to the bit-by-bit AND. The result of the OR comparison, however, is determined by the following rule:

> The result of the comparison is 1 if either bit being compared is a 1, otherwise the result is a 0.

Figure 13–2 illustrates an OR operation. As shown in the figure, when either of the two bits being compared is a 1, the result is a 1, otherwise the result is a 0. As with all bit operations, the result of each comparison is, of course, independent of any other comparison.

Program 13-3 illustrates an OR operation, using the octal values of the operands illustrated in Figure 13–2.

Program 13-3

```
#include <stdio.h>
main( )
{
  int op1 = 0325, op2 = 0263;

  printf("%o ORed with %o is %o",op1, op2, op1 | op2);
}
```

Program 13-3 produces the following output:

```
325 ORed with 263 is 367
```

The result of ORing the octal numbers 325 and 263 is the octal number 367. The binary equivalent of 367 is 1 1 1 1 0 1 1 1, which is the result of the OR operation illustrated in Figure 13–2.

Inclusive OR operations are extremely useful in forcing selected bits to take on a 1 value or for passing through other bit values unchanged. This is a direct result of the fact that ORing any bit (1 or 0) with a 1 forces the resulting bit to be a 1, while ORing any bit (1 or 0) with a 0 leaves the original bit unchanged. For

```
  1 0 1 1 0 0 1 1
| 1 1 0 1 0 1 0 1
-----------------
  1 1 1 1 0 1 1 1
```

FIGURE 13–2 A Sample OR Operation

example, assume that the variable op1 has the arbitrary bit pattern x x x x x x x x, where each x can be either 1 or 0, independent of any other x in the number. The result of ORing this binary number with the binary number 1 1 1 1 0 0 0 0 is:

```
op1 =    x x x x x x x x
op2 =    1 1 1 1 0 0 0 0
         ---------------
Result = 1 1 1 1 x x x x
```

As can be seen from this example, the ones in op2 force the resulting bits to 1, while the zeros in op2 filter, or pass, the respective bits in op1 through with no change in their values. Thus, using an OR operation a masking operation similar to an AND operation can be produced, except the masked bits are set to ones rather than cleared to zeros. Another way of looking at this is to say that ORing with a zero has the same effect as ANDing with a one.

Program 13-4 uses this masking property to convert uppercase letters in a

Program 13-4

```
#include <stdio.h>
main( )
{
  char word[81];       /* enough storage for a complete line */
  void lower(char *); /* function prototype */

  printf("Enter a string of both upper and lowercase letters:\n");
  gets(word);
  printf("\nThe string of letters just entered is:\n");
  puts(word);
  lower(word);
  printf("\nThis string, in lowercase letters is:\n");
  puts(word);
}
void lower(char *word)
{
  while (*word != '\0')
    *word++ |= 0X20;
}
```

word into their respective lowercase form, assuming the letters are stored using the ASCII code. The algorithm for converting letters is similar to that used in Program 13-2. It converts uppercase letters into their lowercase form by forcing the sixth bit in each letter to a one. This is accomplished in Program 13-4 by masking the letter's code with the binary value `00100000`, which has the hexadecimal value 20.

A sample run using Program 13-4 follows:

```
Enter a string of both upper and lowercase letters:
abcdefgHIJKLMNOPqrstuvwxyz

The string of letters just entered is:
abcdefgHIJKLMNOPqrstuvwxyz

This string in lowercase letters is:
abcdefghijklmnopqrstuvwxyz
```

Notice that the uppercase letters are converted to lowercase form, while lowercase letters are unaltered. This is because bit 6 of all lowercase letters is one to begin with, so that forcing this bit to one using the mask has no effect. Only when bit 6 is a zero, as it is for uppercase letters, is the input character altered.

13.3 The Exclusive OR Operator

The exclusive OR operator, `^`, performs a bit-by-bit comparison of its two operands. The result of the comparison is determined by the following rule:

> The result of the comparison is 1 if one and only one of the bits being compared is a 1, otherwise the result is 0.

Figure 13–3 illustrates an exclusive OR operation. As shown in the figure, when both bits being compared are the same value (both 1 or both 0), the result is a zero. Only when both bits have different values (one bit a 1 and the other a 0) is the result a 1. Again, each pair or bit comparison is independent of any other bit comparison.

An exclusive OR operation can be used to create the opposite value, or complement, of any individual bit in a variable. This is a direct result of the fact that exclusive ORing any bit (1 or 0) with a 1 forces the resulting bit to be of the opposite value of its original state, while exclusive ORing any bit (1 or 0) with a 0 leaves the original bit unchanged. For example, assume that the variable `op1` has

FIGURE 13–3 A Sample Exclusive OR Operation

```
  1 0 1 1 0 0 1 1
^ 1 1 0 1 0 1 0 1
-----------------
  0 1 1 0 0 1 1 0
```

the arbitrary bit pattern x x x x x x x x, where each x can be either 1 or 0, independent of any other x in the number. Using the notation that $\bar{x}$ is the complement (opposite) value of x, the result of exclusive ORing this binary number with the binary number 0 1 0 1 0 1 0 1 is:

```
op1 =    x x x x x x x x
op2 =    0 1 0 1 0 1 0 1
         ---------------
Result = x̄ x x̄ x x̄ x x̄ x
```

As can be seen from this example, the ones in op2 force the resulting bits to be the complement of their original bit values, while the zeros in op2 filter, or pass, the respective bits in op1 through with no change in their values.

Many encryption methods use the exclusive OR operation to code data. For example, the string Hello there world! initially used in Program 1-1 can be encrypted by exclusive ORing each character in the string with a mask value of 52. The choice of the mask value, which is referred to as the encryption key, is arbitrary. Any key value can be used.

Program 13-5 uses an encryption key of 52 to code a user-entered message.

Program 13-5

```
#include <stdio.h>
main( )
{
  char message[81];       /* enough storage for a complete line */
  void encrypt(char *);   /* function prototype*/

  printf("Enter a sentence:\n");
  gets(message);
  printf("\nThe sentence just entered is:\n");
  puts(message);
  encrypt(message);
  printf("\nThe encrypted version of this sentence is:\n");
  puts(message);
}
void encrypt(char *message)
{
  while (*message != '\0')
    *message++ ^= 52;
}
```

Following is a sample run using Program 13-5.

```
Enter a sentence:
Good morning
```

```
The sentence just entered is:
Good morning

The encrypted version of this sentence is:
s[[P Y[FZ]ZS
```

Note that the encrypted version appears to have fewer characters. This is due to the non-printable codes contained within the encrypted version. Decoding an encrypted message requires exclusive ORing the coded message using the original encryption key, which is left as a homework exercise.

13.4 The Complement Operator

The complement operator, ~, is a unary operator that changes each 1 bit in its operand to 0 and each 0 bit to 1. For example, if the variable `op1` contains the binary number `11001010`, `~op1` replaces this binary number with the number `00110101`. The complement operator is used to force any bit in an operand to zero, independent of the actual number of bits used to store the number. For example, the statement

```
op1 = op1 & ~07;
```

or its shorter form,

```
op1 &= ~07;
```

both set the last three bits of `op1` to zero, regardless of how `op1` is stored within the computer. Either of these two statements can, of course, be replaced by ANDing the last three bits of `op1` with zeros, if the number of bits used to store `op1` is known. In a computer that uses 16 bits to store integers, the appropriate AND operation is:

```
op1 = op1 & 0177770;
```

For a computer that uses 32 bits to store integers, the above AND sets the left-most or higher order 16 bits to zero also, which is an unintended result. The correct statement for 32 bits is:

```
op1 = op1 & 037777777770;
```

Using the complement operator in this situation frees the programmer from having to determine the storage size of the operand and, more importantly, makes the program portable between machines using different integer storage sizes.

13.5 Different-Size Data Items

When the bit operators `&`, `|`, and `^` are used with operands of different sizes, the shorter operand is always increased in bit size to match the size of the larger operand. Figure 13–4 illustrates the extension of a 16-bit unsigned integer into a 32-bit number.

As the figure shows, the additional bits are added to the left of the original number and filled with zeros. This is the equivalent of adding leading zeros to the number, which has no effect on the number's value.

When extending signed numbers, the original leftmost bit is reproduced in the additional bits that are added to the number. As illustrated in Figure 13–5, if

FIGURE 13–4 Extending 16-Bit Unsigned Data to 32 Bits

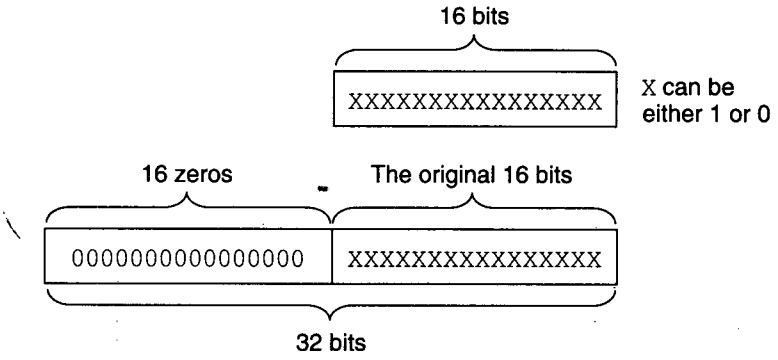
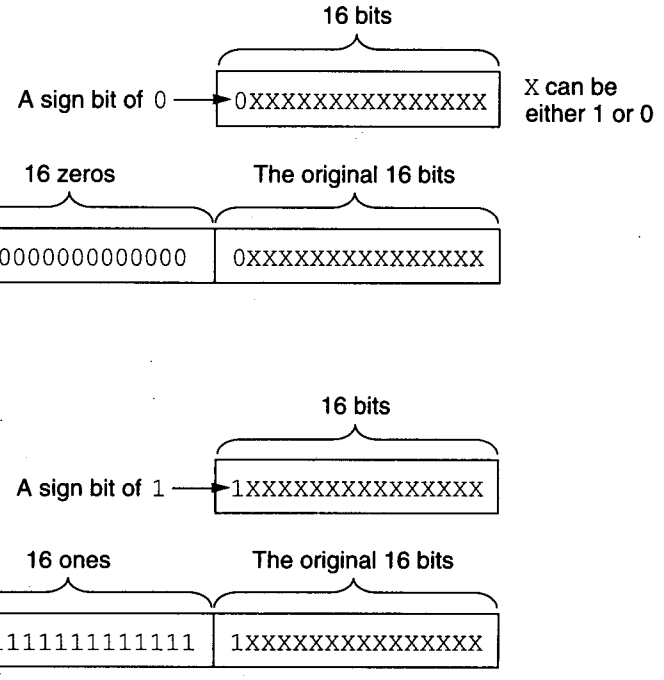


FIGURE 13–5 Extending 16-Bit Signed Data to 32 Bits



the original leftmost bit is 0, corresponding to a positive number, 0 is placed in each of the additional bit positions. If the leftmost bit is 1, which corresponds to a negative number, 1 is placed in the additional bit positions. In either case, the resulting binary number has the same sign and magnitude as the original number.

13.6 The Shift Operators

The left shift operator, <<, causes the bits in an operand to be shifted to the left by a given amount. For example, the statement

```
op1 = op1 << 4;
```

causes the bits in `op1` to be shifted four bits to the left, filling any vacated bits with a zero. Figure 13–6 illustrates the effect of shifting the binary number `1111100010101011` to the left by four bit positions.

For unsigned integers, each left shift corresponds to multiplication by 2. This is also true for signed numbers using two's complement representation, as long as the leftmost bit does not switch values. Since a change in the leftmost bit of a two's complement number represents a change in both the sign and magnitude represented by the bit, such a shift does not represent a simple multiplication by 2.

The right shift operator, >>, causes the bits in an operand to be shifted to the right by a given amount. For example, the statement

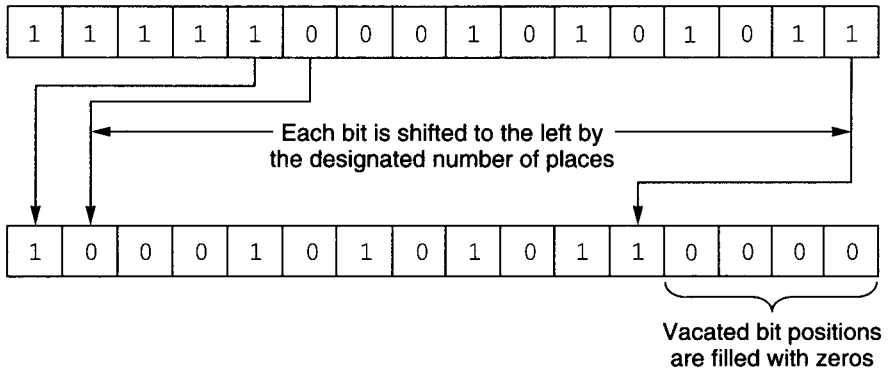
```
op2 = op1 >> 3;
```

causes the bits in `op1` to be shifted to the right by three bit positions. Figure 13–7a illustrates the right shift of the unsigned binary number `1111100010101011` by three bit positions. As illustrated, the three rightmost bits are shifted "off the end" and are lost.

For unsigned numbers, the leftmost bit is not used as a sign bit. For this type of number, the vacated leftmost bits are always filled with zeros. This is the case that is illustrated in Figure 13–7a.

For signed numbers, what is filled in the vacated bits depends on the computer. Most computers reproduce the original sign bit of the number. Figure 13–7b

FIGURE 13–6 An Example of a Left Shift



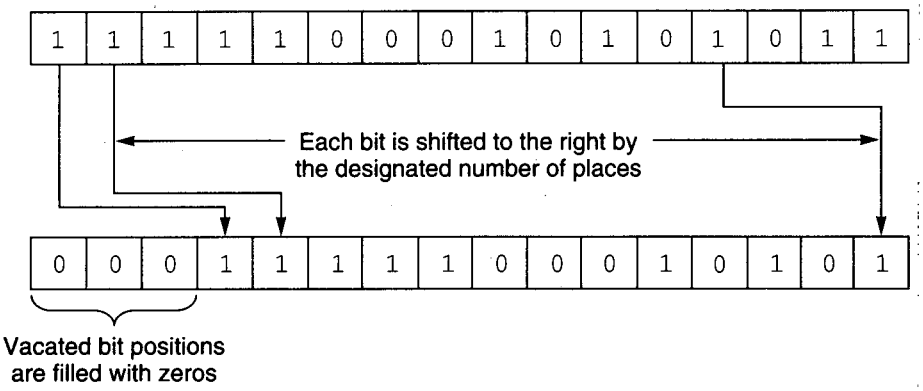


FIGURE 13–7a An Unsigned Arithmetic Right Shift

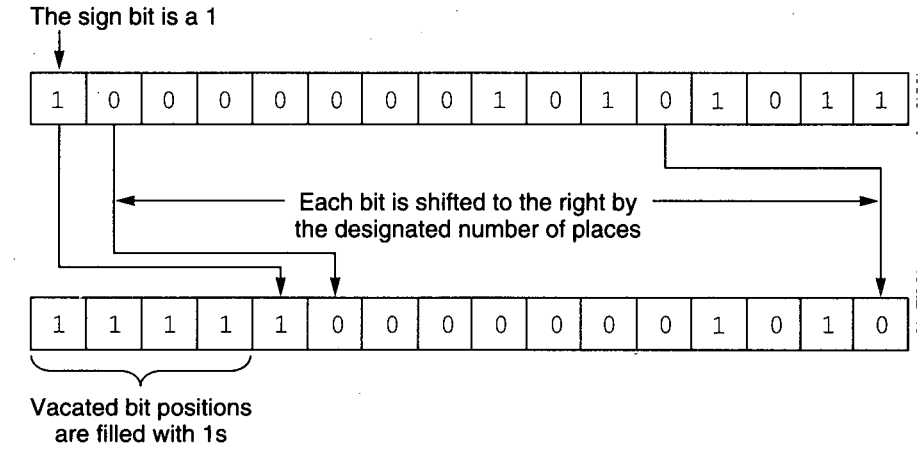


FIGURE 13–7b The Right Shift of a Negative Binary Number

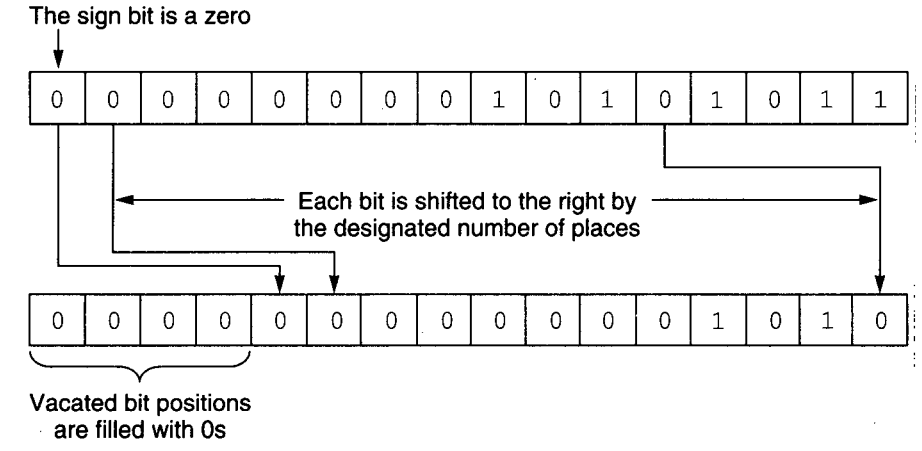


FIGURE 13–7c The Right Shift of a Positive Binary Number

illustrates the right shift of a negative binary number by four bit positions, where the sign bit is reproduced in the vacated bits. Figure 13–7c illustrates the equivalent right shift of a positive signed binary number.

The type of fill illustrated in Figures 13–7b and c, where the sign bit is reproduced in vacated bit positions, is called an arithmetic right shift. In an arithmetic right shift, each single shift to the right corresponds to a division by 2.

Instead of reproducing the sign bit in right-shifted signed numbers, some computers automatically fill the vacated bits with zeros. This type of shift is called a logical shift. For positive signed numbers, where the leftmost bit is zero, both arithmetic and logical right shifts produce the same result. The results of these two shifts are different only when negative numbers are involved.

Exercises for Chapter 13

1. Determine the results of the following operations:

```
a.   11001010    b.   11001010    c.   11001010
   & 10100101       | 10100101       ^ 10100101
   ----------       ----------       ----------
```

2. Write the octal representations of the binary numbers given in Exercise 1.

3. Determine the octal results of the following operations, assuming unsigned numbers:
   a. the octal number 0157 shifted left by one bit position
   b. the octal number 0701 shifted left by two bit positions
   c. the octal number 0673 shifted right by two bit positions
   d. the octal number 067 shifted right by three bit positions

4. Repeat Exercise 3 assuming that the numbers are treated as signed values.

5*a.* Assume that the arbitrary bit pattern xxxxxxxx, where each x can represent either 1 or 0, is stored in the integer variable `flag`. Determine the octal value of a mask that can be ANDed with the bit pattern to reproduce the third and fourth bits of `flag` and set all other bits to zero. The rightmost bit in `flag` is considered bit zero.
   b. Determine the octal value of a mask that can be inclusively ORed with the bit pattern in `flag` to reproduce the third and fourth bits of `flag` and set all other bits to one. Again, consider the rightmost bit in `flag` to be bit zero.
   c. Determine the octal value of a mask that can be used to complement the values of the third and fourth bits of `flag` and leave all other bits unchanged. Determine the bit operation that should be used with the mask value to produce the desired result.

6*a.* Write the two's complement form of the decimal number −1, using eight bits. (*Hint:* Refer to Section 1.7 for a review of two's complement numbers.)
   b. Repeat Exercise 6a using 16 bits to represent the decimal number −1 and compare your answer to your previous answer. Could the 16-bit version have been obtained by sign-extending the 8-bit version?

7. As was noted in the text, Program 13-2 has no effect on uppercase letters. Using the ASCII codes listed in Appendix F, determine what other characters would be unaffected by Program 13-2.

8. Modify Program 13-2 so that a complete sentence can be read in and converted to lowercase values. (*Hint:* When a space is masked by Program 13-2, the resulting character is `\0`, which terminates the output.)

9. Modify Program 13-4 to allow a complete sentence to be input and converted to uppercase letters. Make sure that your program does not alter any other characters or symbols entered.

10. Modify Program 13-5 to permit the encryption key to be a user-entered input value.

11. Modify Program 13-5 to have its output written to a file named `coded.dat`.

12. Write a C program that reads the encrypted file produced by the program written for Exercise 10, decodes the file, and prints the decoded values.

13. Write a C program that displays the first eight bits of each character value input into a variable named `ch`. (*Hint:* Assuming each character is stored using eight bits, start by using the hexadecimal mask 80, which corresponds to the binary number `10000000`. If the result of the masking operation is a zero, display a zero; else display a one. Then shift the mask one place to the right to examine the next bit, and so on until all bits in the variable `ch` have been processed.)

14. Write a C program that reverses the bits in an integer variable named `okay` and stores the reversed bits in the variable named `rev_okay`. For example, if the bit pattern `11100101`, corresponding to the octal number 0345, is assigned to `okay`, the bit pattern `10100111`, corresponding to the octal number 0247, should be produced and stored in `rev_okay`.

13.7 Chapter Summary

1. Individual bits of character and integer variables and constants can be manipulated using C's bit operators. These are the AND, inclusive OR, exclusive OR, one's complement, left shift, and right shift operators.
2. The AND and inclusive OR operators are useful in creating masks. These masks can be used to pass or eliminate individual bits from the selected operand. The exclusive OR operator is useful in complementing an operand's bits.
3. When the AND and OR operators are used with operands of different sizes, the shorter operand is always increased in bit size to match the size of the larger operand.
4. The shift operators produce different results depending on whether the operand is a signed or an unsigned value.

Additional Capabilities

Chapter Fourteen

14.1 Expressions Revisited
14.2 User-Specified Data Types
14.3 Defining Macros
14.4 Command Line Arguments
14.5 The `goto` Statement
14.6 Chapter Summary

Previous chapters presented C's basic capabilities and structure. The variations on these capabilities, which are almost endless, are a source of delight to many programmers who continuously find new possibilities of expression using variations of the basic language building blocks. This chapter presents some of these additional capabilities.

14.1 Expressions Revisited

One of the most common pitfalls in C results from misunderstanding the full implications of an expression. Recall that an expression is any combination of operands and operators that yields a result. This definition is extremely broad and more encompassing than is initially apparent. For example, all of the following are valid C expressions:

```
a + 5
a = b
a == b
a = b = c = 6
flag = a == b
```

Assuming that the variables are suitably declared, each of the above expressions yields a result. Program 14-1 uses the `printf( )` function to display the value of the first three expressions for specific initial values of the variables `a` and `b`.

Program 14-1

```
#include <stdio.h>
main( )
{
  int a = 7, b = 10;

  printf("\nThe value of the first expression is %d", a + 5);
  printf("\nThe value of the second expression is %d", a == b);
  printf("\nThe value of the third expression is %d", a = b );
}
```

The display produced by Program 14-1 is:

```
The value of the first expression is 12
The value of the second expression is 0
The value of the third expression is 10
```

As the output of Program 14-1 illustrates, each expression, by itself, has a value associated with it. The value of the first expression is the sum of the vari-

able a plus 5, which is 12. The value of the second expression is zero, since a is not equal to b, and a false condition is represented in C with a value of zero. If the values in a and b had been the same, the conditional expression a == b would be true and would have a value of 1. The value of the third expression is 10, which is also assigned to the variable a.

In this section we will review the rules for evaluating expressions with multiple operators and "mixed" operands of different data types. We will also introduce a new expression type and C operator.

Expressions containing multiple operators are always evaluated by the priority, or precedence, of each operator. Table E–1 in Appendix E lists the relative priority of each C operator and its associativity.

Even when the order of evaluation is known, expressions with multiple operators can still produce unusual and undesired results, remaining a potential trap for the unwary. For example, consider the statement

```
flag = a == b;
```

Consulting Table E–1 we see that the == operator has a higher precedence than the = operator. Therefore, a is first compared to b. If a is equal to b, the result of the expression a == b is 1, otherwise it is 0. The value of this expression is then assigned to flag. Thus, the variable flag will have either a value of 1 or a value of 0 after the statement is executed. A problem arises if the expression is inadvertently typed as flag = a = b. Here, the value of b is first assigned to a, which is then assigned to flag. Because of the mistake of typing an equal operator instead of the comparison operator, flag is assigned the value of b, rather than the 1 or 0 that was intended.

The real problem with the statement flag = a == b; is that it has been used in place of the more obvious and complete statement

```
if (a == b)
  flag = 1;
else
  flag = 0;
```

Although the same error can be made in substituting an equal operator for the comparison operator in this statement, the error can be detected more easily than in the more obscure expression flag = a == b.

Because of the generality of C expressions and the fact that most sequences of operands connected by operators can be evaluated to produce a result (including an unintended one), it is extremely important to be careful in creating expressions. To avoid undesired results and to make program checking, testing, and debugging easier, keep expressions as simple and as uncomplicated as possible. Generally expressions using arithmetic operators (+, -, *, /, %, etc.) should not be mixed with expressions using relational and logical operators (==, <, >, &&, ||, etc.), which, in turn, should not be mixed with expressions using bit operators (&, |, etc.).

Finally, although Table E–1 appears to be all-inclusive, it is not. In particular, the order of evaluations for operands is not specified, as it is not specified in most computer languages. For example, in the expression a + b it is not known which operand is accessed first. Generally this is not a problem because the order of operand access doesn't affect the result of the expression. However, in expressions such as:

```
(val[i]) + (i++)
```

the order of access is important. Here the subscript may be either the old or the new value of `i`, depending on which operand is accessed first.

Similarly, the order of evaluation of function arguments is not specified in C. Thus, the function call

```
printf("%d %d", i, i++);
```

may result in the same number being printed twice if the second argument is evaluated before the first argument.

Expressions that depend on the order of operand access should always be avoided, because they can produce different results on different computers. Such expressions can always be replaced with temporary variables that explicitly define the desired evaluation order. For example, the statements

```
n = ++i;
printf("%d %d", i, n);
```

clearly indicate the values that are passed to the `printf( )` function.

Conditional Expressions

In addition to expressions formed with the arithmetic, relational, logical, and bit operators, C provides a conditional expression. A conditional expression uses the conditional operator, `?:`. It provides an alternate way of expressing a simple `if-else` statement.

The general form of a conditional expression is:

```
expression1 ? expression2 : expression3
```

If the value of `expression1` is nonzero (true), `expression2` is evaluated, otherwise `expression3` is evaluated. The value for the complete conditional expression is the value of either `expression2` or `expression3`, depending on which expression was evaluated. As always, the value of the expression may be assigned to a variable.

Conditional expressions are most useful in replacing simple `if-else` statements. For example, the `if-else` statement

```
if ( hours > 40)
  rate = .045;
else
  rate = .02;
```

can be replaced with the one-line statement

```
rate = (hours > 40) ? .045 : .02;
```

Here, the complete conditional expression

```
(hours > 40) ? .045 : .02
```

is evaluated before any assignment is made to `rate`, because the conditional

operator, `?:`, has a higher precedence than the assignment operator. Within the conditional expression, the expression `hours > 40` is evaluated first. If this expression has a nonzero value, which is equivalent to a logical true value, the value of the complete conditional expression is set to .045; otherwise the conditional expression has a value of .02. Finally, the value of the conditional expression, either .045 or .02, is assigned to the variable `rate`.

The conditional operator, `?:`, is unique in C in that it is a ternary operator. This means that the operator connects three operands. The first operand is always evaluated first. It is usually a conditional expression that uses the logical operators. The next two operands are any other valid expressions, which can be single constants, variables, or more general expressions. The complete conditional expression consists of all three operands connected by the conditional operator symbols, `?` and `:`.

Conditional expressions are only useful in replacing `if-else` statements when the expressions in the equivalent `if-else` statements are not long or complicated. For example, the statement

```
max_val = a > b ? a : b;
```

is a one-line statement that assigns the maximum value of the variables `a` and `b` to `max_val`. A longer, equivalent form of this statement is:

```
if (a > b)
  max_val = a;
else
  max_val = b;
```

Because of the length of the expressions involved, a conditional expression would not be useful in replacing the following `if-else` statement:

```
if (amount > 20000)
  taxes = .025(amount - 20000) + 400;
else
  taxes = .02 * amount;
```

Exercises 14.1

1. Evaluate the following expressions. Assume that all variables are integers and that `a` has a value of 2, `b` has a value of 3, `c` has a value of 4, and `d` has a value of 5 before each expression is evaluated.

a. `b + 3`
b. `a = b + 3`
c. `a = b = c = d + 4 * a`
d. `flag = a >= b`
e. `flag = a == b || c == d`
f. `d == a = b`
g. `a + b > 20`
h. `num = a + b > 20`
i. `a || b`
j. `num = a || b`

2. Which of the expressions in Exercise 1 should not be included in a program? Why?

3. Rewrite the statement `a = b = c = amount * rate;` as a series of three individual assignment statements.

4. Rewrite the following statements as `if-else` statements:
   a. `flag = a >= b;`
   b. `flag = a == b || c == d;`

5. Rewrite the statements in Exercise 4 using conditional expressions.

6. Rewrite each of the following `if-else` statements using a conditional expression:

a.
```
if (a < b);
    min_val = a;
  else
    min_val = b;
```

b.
```
if (num < 0)
    sign = -1;
  else
    sign = 1;
```

c.
```
if (flag == 1)
    val = num;
  else
    val = num * 'num;
```

d.
```
if (credit == plus)
    rate = prime;
  else
    rate = prime + delta;
```

e.
```
if (!bond)
    cou = .075;
  else
    cou = 1.1;
```

14.2 User-Specified Data Types

In this section we present two user-specified data types. The first permits a user to create new data types. Since the creation of a data type requires the programmer to specifically list or enumerate the values appropriate to the data type, these data types are referred to as enumerated data types. The second capability allows the programmer to create new names for existing data types.

Enumerated Data Types

An enumerated data type is a user-created data type in which the values appropriate to the data type are specified in a user-defined list. Such data types are identified by the reserved word `enum` followed by an optional, user-selected name for the data type and a listing of acceptable values for the data type. Consider the following user-specified data types:

```
enum flag {true, false};
enum time {am, pm};
enum day {mon, tue, wed, thr, fri, sat, sun};
enum color {red, green, yellow};
```

The first user-specified data type is the type `flag`. Any variable subsequently declared to be of this type can take on only a value of true or false. The second statement creates a data type named `time`. Any variable subsequently declared to be of type time can take on only a value of `am` or `pm`. Similarly, the third and fourth statements create the data types `day` and `color`, respectively, and list the valid values for variables of these two types. For example, the statement

```
enum day a,b,c;
```

declares the variables `a`, `b`, and `c` to be of type `day`, and is consistent with the declaration of variables using standard C data types such as `char`, `int`, `float`, or `double`. Once variables have been declared as enumerated types, they may be assigned values or compared to variables or values appropriate to their type. This again is consistent with standard variable operations. For example, for the variables `a`, `b`, and `c` declared above, the following statements are valid:

```
a = red;
b = a;
if (c == yellow) printf("\nThe color is yellow");
```

Internally, the acceptable values for each enumerated data type are ordered and assigned sequential integer values beginning with 0. For example, for the values of the user-defined type `color`, the correspondences created by the C compiler are that `red` is equivalent to 0, `green` is equivalent to 1, and `yellow` is equivalent to 2. The equivalent numbers are required when inputting values using `scanf( )` or printing values using `printf( )`. Program 14-2 illustrates a user-defined data type.

Program 14-2

```
#include <stdio.h>
main( )
{
  enum color {red,green,yellow};
  enum color crayon = red;  /* crayon is declared to be of type  */
                            /* color and initialized to red      */
  printf("\nThe color is %d", crayon);
  printf("\nEnter in a value: ");
  scanf("%d", &crayon);
  if (crayon == red)
    printf("The crayon is red.");
  else if (crayon == green)
    printf("The crayon is green.");
  else if (crayon == yellow)
    printf("The crayon is yellow.");
  else
    printf("The color is not defined.");
}
```

A sample run of Program 14-2 produced the following output:

```
The color is 0
Enter a value: 2
The crayon is yellow.
```

As illustrated in Program 14-2, expressions containing variables declared as user-defined data types must be consistent with the values specifically listed for the type. Although a switch statement would be more appropriate in Program 14-2, the expressions in the `if-else` statement better highlight the use of enumerated values. Program 14-2 also shows that the initialization of a user-specified data type variable is identical to the initialization of standard data type variables. For input and output purposes, however, the equivlent integer value assigned by the C compiler to each enumerated value must be used in place of the actual data type value. This is also seen in the program.

In order to assign equivalent integers to each user-specified value, the C compiler retains the order of the values as they are listed in the enumeration. A side effect of this ordering is that expressions can be constructed using relational and logical operators. For example, for the data type `color` created in Program 14-2, expressions such as `crayon < yellow` and `red < green` are both valid.

The numerical value assigned by the compiler to enumerated values can be altered by direct assignment when a data type is created. For example, the definition

```
enum color (red,green = 7, yellow);
```

causes the compiler to associate the value `red` with the integer 0 and the value `green` with the integer 7. Altering the integer associated with the value `green` causes all subsequent integer assignments to be altered too; thus, the value `yellow` is associated with the integer 8. If any other values were listed after `yellow`, they would be associated with the integers 9, 10, 11, and so on, unless another alteration was made.

Naming a user-defined data type is similar to naming a template for structures. Just as a template name can be omitted when defining a structure by declaring the structure directly, the same can be done with user-defined data types. For example, the declaration `enum {red,green,yellow} crayon;` defines `crayon` to be an enumerated variable with the valid values of `red`, `green`, and `yellow`.

Scope rules applicable to the standard C data types also apply to enumerated data types. For example, placing the statement `enum color {red, green, yellow};` before the `main( )` function in Program 14-2 would make the data type named `color` global and available for any other function in the file.

Finally, since there is a one-to-one correspondence between integers and user-defined data types, the cast operator can either coerce integers into a user-specified data value or coerce a user-specified value into its equivalent integer. Assuming that `val` is an integer variable with a value of 1, and `color` has been declared as in Program 14-2, the expression `(enum color) val` has a value of `green` and the expression `(int) yellow` has a value of 2. The compiler will not warn you, however, if a cast to a nonexistent value is attempted.

The typedef Statement

In addition to creating new data types, C allows both standard and user-defined data types to be renamed using `typedef` statements. The statement

```
typedef float REAL;
```

makes the name `REAL` a synonym for `float`. The name `REAL` can now be used in place of the term `float` anywhere in the program after the synonym has been declared. For example, the definition

```
REAL val;
```

is equivalent to the definition

```
float val;
```

The `typedef` statement does not create a new data type; it creates a new name for an existing data type. Using uppercase names in `typedef` statements is not mandatory. It is done simply to alert the programmer to a user-specified name, similar to uppercase names in `#define` statements. In fact, the equivalence produced by a `typedef` statement can frequently be produced equally well by a `#define` statement. The difference between the two, however, is that `typedef` statements are processed directly by the compiler while `#define` statements are processed by the preprocessor. Compiler processing of `typedef` statements allows for text replacements that are not possible with the preprocessor. For example, the statement

```
typedef float REAL;
```

actually specifies that `REAL` is a placeholder that will be replaced with another variable name. A subsequent declaration such as

```
REAL val;
```

has the effect of substituting the variable named `val` for the placeholder named `REAL` in the terms following the word `typedef`. Substituting `val` for `REAL` in the `typedef` statement and retaining all terms after the reserved word `typedef` results in the equivalent declaration `float val;`.

Once the mechanics of the replacement are understood, more useful equivalences can be constructed. Consider the statement

```
typedef int ARRAY[100];
```

Here, the name `ARRAY` is actually a placeholder for any subsequently defined variables. Thus, a statement such as `ARRAY first, second;` is equivalent to the two definitions

```
int first[100];

int second[100];
```

Each of these definitions is obtained by replacing the name ARRAY with the variable names first and second in the terms following the reserved word typedef.

As another example, consider the following statement:

```
typedef struct
        {
           char name[20];
           int id_num;
        } EMP_REC;
```

Here EMP_REC is a convenient placeholder for any subsequent variable. For example, the declaration EMP_REC employee[75]; is equivalent to the declaration

```
struct
{
   char name[20];
   int id_num;
} employee[75];
```

This last declaration is obtained by directly substituting the term employee[75] in place of the word EMP_REC in the terms following the word typedef in the original typedef statement.

14.3 Defining Macros

In its simplest form, the #define preprocessor is used to equate constants and operators to symbolic names. For example, the statement

```
#define SALESTAX .05
```

equates the symbolic name SALESTAX to the number .05. When SALESTAX is used in any subsequent statement or expression the equivalent value of .05 is substituted for the symbolic name. The substitutions are made by the C preprocessor just prior to program compilation.

C places no restrictions on the equivalences that can be established with the #define statement. The symbolic name following the #define designation can be equated to any text and can even include arguments. For example, all of the following are valid equivalences:

```
#define  PI       3.1416
#define  TIMES    *
#define  EQUALS   =
#define  FORMAT   "The answer is %f"
```

The use of these equivalence statements is illustrated in Program 14-3.

Program 14-3

```
#include <stdio.h>
#define  PI        3.1416
#define  TIMES     *
#define  EQUALS    =
#define  FORMAT    "The answer is %f"
main( )
{
  float circum, radius = 6.3;

  circum EQUALS 2.0 TIMES PI TIMES radius;
  printf(FORMAT,circum);
}
```

Before Program 14-3 is compiled, the preprocessor directly substitutes the equivalent operator, constant, variable, or text in place of each subsequent occurrence of the symbolic name.

In addition to using `#define` preprocessor statements for simple equivalences, as in Program 14-3, these statements can also be used to equate symbolic names to either partial or complete expressions. When the equivalent text consists of more than a single value, operator, or variable, the symbolic name is referred to as a *macro*, and the substitution of the text in place of the symbolic name is called a *macro expansion* or *macro substitution*. The word macro refers to the direct, in-line expansion of one word into many words. For example, the equivalence established by the statement

```
#define CONVERT   2.0 * 3.1416
```

enables us to write the statement

```
circum = CONVERT * radius;
```

When this statement is encountered by the preprocessor, the symbolic name `CONVERT` is replaced by the equivalent text `2.0 * 3.1416`. The compiler always receives the expanded version after the text has been inserted in place of the symbolic name by the preprocessor. This direct substitution of the text for `CONVERT` occurs in every place that `CONVERT` is encountered after it has been defined. This allows a previously defined symbolic name to be used in subsequent symbolic definitions. For example, the definition for `CONVERT` could have been established using the following set of `#define` statements:

```
#define PI        3.1416
#define CONVERT   2.0 * PI
```

Since `PI` is made equivalent to the constant `3.1416` in the first `#define` statement, it can be used legitimately in any following `#define` statements.

In addition to using `#define` statements for straight text substitutions, these statements can also be used to define equivalences that use arguments. For example, in the equivalence statement

```
#define SQUARE(x)     x * x
```

`x` is an argument. Here, `SQUARE(x)` is a true macro that is expanded into the expression `x * x`, where `x` is itself replaced by the variable or constant used when the macro is utilized. For example, the statement

```
y = SQUARE(num);
```

is expanded into the statement

```
y = num * num;
```

The advantage of using a macro such as `SQUARE(x)` is that since the data type of the argument is not specified, the macro can be used with any data type argument. If `num`, for example, is an integer variable, the expression `num * num` produces an integer value. Similarly, if `num` is a double precision variable, the `SQUARE(x)` macro produces a double precision value. This is a direct result of the text substitution procedure used in expanding the macro and is an advantage of making `SQUARE(x)` a macro rather than a function.

Take care when defining macros with arguments. For example, in the definition of `SQUARE(x)`, there must be no space between the symbolic name `SQUARE` and the left parenthesis used to enclose the argument. There can, however, be spaces within the parentheses if more than one argument is used.

Additionally, since the expansion of a macro involves direct text substitution, unintended results may occur if you do not use macros carefully. For example, the assignment statement

```
val = SQUARE(num1 + num1);
```

does not assign the value of `(num1 + num2)`^{2} to `val`. Rather, the expansion of `SQUARE(num1 + num2)` results in the equivalent statement

```
val =  num1 + num2 * num1 + num2;
```

This statement results from the direct text substitution of the term `num1 + num2` for the argument `x` in the expression `x * x` that is produced by the preprocessor.

To avoid unintended results, always place parentheses around all macro arguments wherever they appear in the macro. For example, the definition

```
#define SQUARE(x)   (x) * (x)
```

ensures that a correct result is produced whenever the macro is invoked. Now the statement

```
val = SQUARE(num1 + num2);
```

is expanded to produce the desired assignment:

```
val = (num1 + num2) * (num1 + num2);
```

Macros are extremely useful when the calculations or expressions they contain are relatively simple and can be kept to one or at most two lines. Larger macro definitions tend to become cumbersome and confusing; they are better written as functions. If necessary, a macro definition can be continued on a new line by typing a backslash character, \, before the RETURN or ENTER key is pressed. The backslash acts as an escape character that causes the preprocessor to treat the RETURN literally and not include it in any subsequent text substitutions.

The advantage of using a macro instead of a function is an increase in execution speed. Since the macro is directly expanded and included in every expression or statement using it, there is no execution time loss due to the call and return procedures required by a function. The disadvantage is the increase in required program memory space when a macro is used repeatedly. Each time a macro is used the complete macro text is reproduced and stored as an integral part of the program. Thus, if the same macro is used in ten places, the final code includes ten copies of the expanded text version of the macro. A function, however, is stored in memory only once. No matter how many times the function is called, the same code is used. The memory space required for one copy of a function used extensively throughout a program can be considerably less than the memory required for storing multiple copies of the same code defined as a macro.

Exercises 14.3

1 *a.* Define a macro named `NEGATE(x)` that produces the negative of its argument.
b. Include the `NEGATE(x)` macro defined in Exercise 1a in a complete C program and run the program to confirm proper operation of the macro for various cases.

2 *a.* Define a macro named `ABS_VAL(x)` that produces the absolute value of its argument.
b. Include the `ABS_VAL(x)` macro defined in Exercise 2a in a complete C program and run the program to confirm proper operation of the macro for various cases.

3 *a.* Define a macro named `CIRCUM(r)` that determines the circumference of a circle of radius *r*. The circumference is determined from the relationship circumference = 2.0 * π* radius, where π equals 3.1416.
b. Include the `CIRCUM(x)` macro defined in Exercise 3a in a complete C program and run the program to confirm proper operation of the macro for various cases.

4 *a.* Define a macro named `MIN(x,y)` that determines the minimum value of its two arguments.
b. Include the `MIN(x,y)` macro defined in Exercise 4a in a complete C program and run the program to confirm proper operation of the macro for various cases.

5 *a.* Define a macro named `MAX(x,y)` that determines the maximum value of its two arguments.
b. Include the `MAX(x,y)` macro defined in Exercise 5a in a complete C program and run the program to confirm proper operation of the macro for various cases.

14.4 Command Line Arguments

Arguments can be passed to any function in a program, including the `main( )` function. In this section we describe the procedures for passing arguments to `main( )` when a program is initially invoked and having `main( )` correctly receive and store the arguments passed to it. Both the sending and receiving sides of the transaction must be considered. Fortunately, the interface for transmitting arguments to a `main( )` function has been standardized in C, so both sending and receiving arguments can be done almost mechanically.

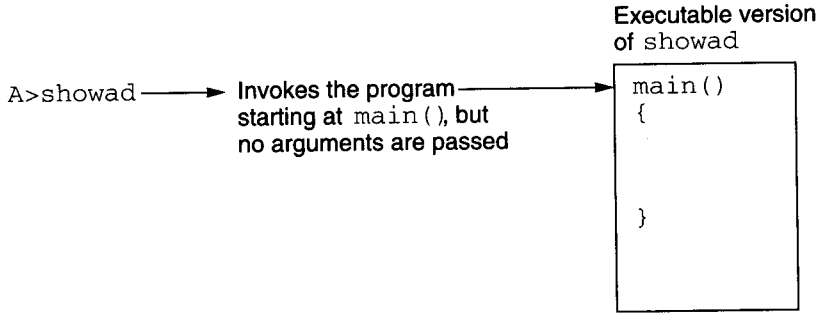
All the programs that have been run so far have been invoked by typing the name of the executable version of the program after the operating system prompt is displayed. The command line for these programs consists of a single word, which is the name of the program. For computers that use the UNIX® Operating System the prompt is usually the `$` symbol and the executable name of the program is `a.out`. For these systems, the simple command line `$a.out` begins program execution of the last compiled source program currently residing in `a.out`.

If you are using a C compiler on an IBM PC, the equivalent operating system prompt is either `A>` or `C>`, and the name of the executable program is typically the same name as the source program with an `.exe` extension rather than a `.c` extension. Assuming that you are using an IBM PC with the `A>` operating system prompt, the complete command line for running an executable program named `showad.exe` is `A> showad`. As illustrated in Figure 14–1, this command line causes the `showad` program to begin execution with its `main( )` function, but no arguments are passed to `main( )`.

Now assume that we want to pass the three separate string arguments `three blind mice` directly into `showad`'s main function. Sending arguments into a `main( )` function is extremely easy. It is accomplished by including the arguments on the command line used to begin program execution. Because the arguments are typed on the command line they are, naturally, called command line arguments. To pass the arguments `three blind mice` directly into the `main( )` function of the `showad` program, we only need to add the desired words after the program name on the command line:

```
A> showad three blind mice
```

FIGURE 14–1 Invoking the showad Program



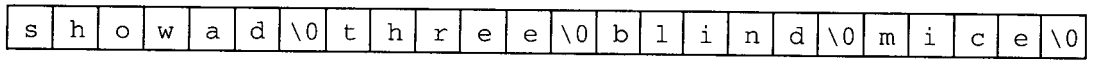


FIGURE 14–2 The Command Line Stored in Memory

Upon encountering the command line `showad three blind mice`, the operating system stores it as a sequence of four strings. Figure 14–2 illustrates the storage of this command line, assuming that each character uses one byte of storage. As shown in the figure, each string terminates with the standard C null character `\0`.

Sending command line arguments to `main( )` is always this simple. The arguments are typed on the command line and the operating system nicely stores them as a sequence of separate strings. We must now handle the receiving side of the transaction and let `main( )` know that arguments are being passed to it.

Arguments passed to `main( )`, like all function arguments, must be declared as part of the function's definition. To standardize argument passing to a `main( )` function, only two items are allowed: a number and an array. The number is an integer variable, which must be named `argc` (short for argument counter), and the array is a one-dimensional list, which must be named `argv` (short for argument values). Figure 14–3 illustrates these two arguments.

The integer passed to `main( )` is the total number of items on the command line. In our example, the value of `argc` passed to `main( )` is four, which includes the name of the program plus the three command line arguments. The one-dimensional list passed to `main( )` is a list of pointers containing the starting storage address of each string typed on the command line, as illustrated in Figure 14–4.

We can now write the complete function definition for `main( )` to receive arguments. Since an integer and an array are passed to `main( )` and C requires that these two items be named `argc` and `argv`, respectively, the first line in `main( )`'s definition must be `main(int argc, char *argv[ ])`.

Because `argc` is an integer, its declaration is `int argc;`. Because `argv` is the name of an array whose elements are addresses that point to where the actual command line arguments are stored, its proper declaration is `char *argv[ ];`. This is nothing more than the declaration of an array of pointers. It is read "`argv` is an array whose elements are pointers to characters." Putting all this together, the full function header line for a `main( )` function that will receive command line arguments is:[1]

```
main(int argc, char *argv[ ])  /* complete main() header line */
```

No matter how many arguments are typed on the command line, `main( )` only needs the two standard pieces of information provided by `argc` and `argv`: the number of items on the command line and the list of starting addresses indicating where each argument is actually stored.

---

[1] In traditional (non-ANSI) C, this header is written as:

```
main(argc,argv)
int argc;
char *argv[ ];
```

Strictly speaking, both ANSI and non-ANSI `main( )` function headerlines should also be preceeded by the keyword `void`, since they return no values. Historically, early C compilers did not have an explicit `void` data-type, the keyword `void` is still frequently omitted from the definition for `main( )`.

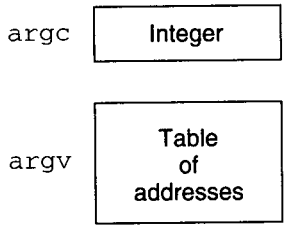


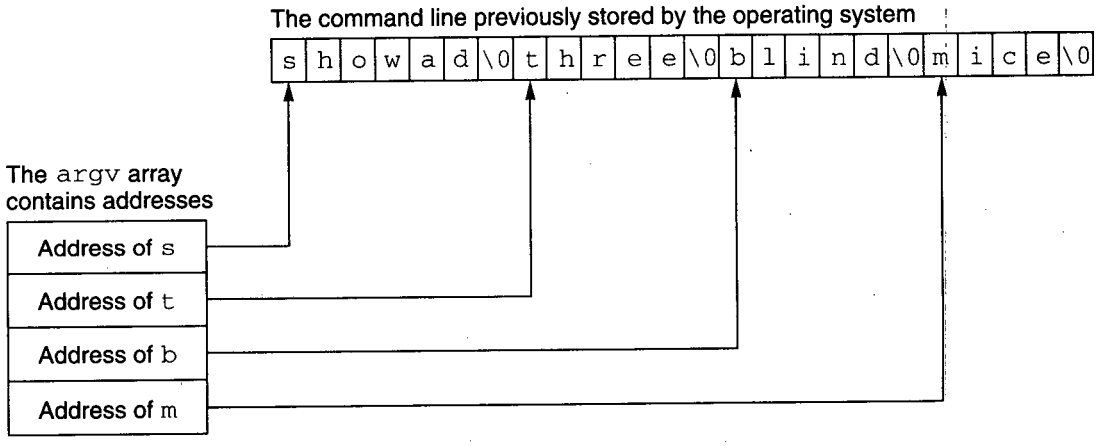
FIGURE 14–3 An Integer and an Array Are Passed to `main( )`

Program 14-4 verifies our description by printing the data actually passed to `main( )`. The variable `argv[i]` used in Program 14-4 contains an address. The notation `*argv[i]` refers to "the character pointed to" by the address in `argv[i]`.

Program 14-4

```
#include <stdio.h>
main(int argc, char *argv[ ])
{
  int i;

  printf("\nThe number of items on the command line is %d\n\n",argc);
  for (i = 0; i < argc; ++i)
  {
    printf("The address stored in argv[%d] is %p\n", i, argv[i]);
    printf("The character pointed to is %c\n\n", *argv[i]);
  }
}
```

FIGURE 14–4 Addresses Are Stored in the `argv` Array



Assuming that the executable version of Program 14-4 is named `showad.exe`, a sample output for the command line `showad three blind mice` is:

```
The number of items on the command line is 4

The address stored in argv[0] is FFE4
The character pointed to is s

The address stored in argv[0] is FFEB
The character pointed to is t

The address stored in argv[0] is FFF1
The character pointed to is b

The address stored in argv[0] is FFF7
The character pointed to is m
```

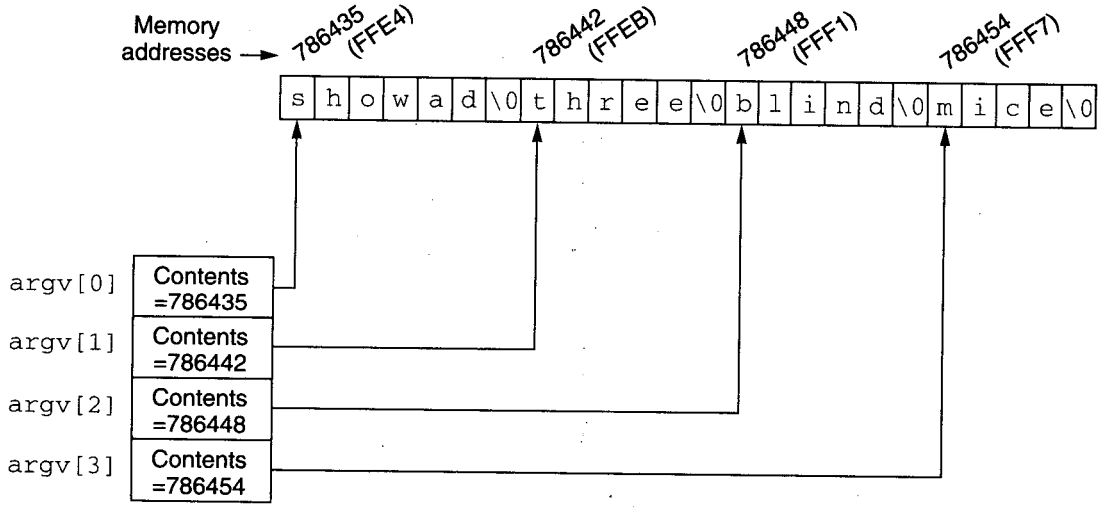
The addresses displayed by Program 14-4 clearly depend on the machine used to run the program. Figure 14–5 illustrates the storage of the command line as displayed by the sample output.[2] As anticipated, the addresses in the `argv` array "point" to the starting characters of each string typed on the command line.

Once command line arguments are passed to a C program, they can be used like any other C strings. Program 14-5 causes its command line arguments to be displayed from within `main( )`.

Assuming that the name of the executable version of Program 14-5 is `a.out`, the output of this program for the command line `a.out three blind mice` is:

```
The following arguments were passed to main( ): three blind mice
```

FIGURE 14–5 The Command Line Stored in Memory



[2] For convenience, the equivalent decimal address values are shown.

Program 14-5

```
/* A program that displays command line arguments */
#include <stdio.h>
main(int argc, char *argv[ ])
{
  int i;

  printf("\nThe following arguments were passed to main(): ");
  for (i = 1; i < argc; ++i)
  printf("%s ", argv[i]);
}
```

Notice that when the addresses in `argv[ ]` are passed to the `printf( )` function in Program 14-5, the strings pointed to by these addresses are displayed. When these same addresses were passed to the `printf( )` function in Program 14-4, the actual values of the addresses were printed. The difference in displays is caused by the `printf( )` function. When a `%s` control sequence is used in `printf( )`, as it is in Program 14-5, it alerts the function that a string will be accessed. `printf( )` then expects the address of the first character in the string; this is exactly what each element in `argv[ ]` supplies.

One final comment about command line arguments: Any argument typed on a command line is considered to be a string. If you want numerical data passed to `main( )`, it is up to you to convert the passed string into its numerical counterpart. This is seldom an issue, however, since most command line arguments are used as flags to pass appropriate processing control signals to an invoked program.

Exercises 14.4

1 *a.* Write a C program that accepts the name of a data file as a command line argument. Have your program open the data file and display its contents, line by line, on the CRT screen.
b. Would the program written for Exercise 1a work correctly for a program file?

2 *a.* Modify the program written for Exercise 1a so that each line displayed is preceded by a line number.
b. Modify the program written for Exercise 2a so that the command line argument `-p` will cause the program to list the contents of the file on the printer attached to your system.

3. Write a C program that accepts a command line argument as the name of a data file. Given the name, your program should display the number of characters in the file. (*Hint:* Use the `fseek( )` and `ftell( )` library functions discussed in Section 8.2.)

4. Write a C program that accepts two integer values as command line arguments. The program should multiply the two values entered and display the result. (*Hint:* The command line must be accepted as string data and converted to numerical values before multiplication.)

14.5 The goto Statement

The goto statement provides an unconditional transfer of control to some other statement in a program. The general form of a goto statement is:

```
goto label;
```

where label is any unique name chosen according to the rules for creating variable names. The label name must appear, followed by a colon, in front of any other statement in the function that contains the goto statement. For example, the following section of code transfers control to the label named err if division by zero is attempted:

```
if (denom == 0.0)
  goto err;
else
  result = num / denom;
        .
        .

err: printf("Error - Attempted Division by Zero";
```

The astute reader will realize that in this case goto provides a cumbersome solution to the problem. It would require a second goto above the printf( ) statement to stop this statement from always being executed. Generally it is much easier either to call an error routine for unusual conditions or to use a break statement if this is necessary.

Theoretically, a goto statement is never required because C's normal structures provide sufficient flexibility to handle all possible flow control requirements. Also, gotos tend to complicate programs. For example, consider the following code:

```
        if (a == 100)
          goto first;
        else
          x = 20;
        goto sec;
first:  x = 50;
  sec:  y = 10;
```

Written without a goto this code is:

```
if (a == 100)
  x = 50;
else
  x = 20;
y = 10;
```

Both sections of code produce the same result; however, the second version clearly is easier to read. It is worthwhile to convince yourself that the two sections of code do, in fact, produce the same result by running the code on your computer. This will let you experience the sense of frustration when working with goto-invaded code.

In C, the goto statement should be used in a limited manner, if at all. The presence of goto statements can rapidly make a program extremely difficult to understand and almost impossible to modify.

14.6 Chapter Summary

1. An expression is any combination of operands and operators that yields a value.
2. A conditional expression provides an alternate way of expressing a simple if-else statement. The general form of a conditional expression is:

```
expression1 ? expression2 : expression3
```

   The equivalent if-else statement for this is:

```
if (expression1)
  expression2;
else
  expression3;
```

3. An enumerated data type is a user-defined scalar data type. The user must select a name and list the acceptable values for the data type. For example, the enumeration

```
enum color {red, green, yellow}
```

   creates a color data type. Any variable may be subsequently declared with this data type and may store one of the acceptable values listed.
4. A typedef statement creates synonyms for both standard and enumerated data types. For example, the statement

```
typedef int WHOLE_NUM;
```

   makes WHOLE_NUM a synonym for int.
5. Using the #define command, complete expressions can be equated to symbolic names. These expressions can include arguments.
6. Arguments passed to main( ) are termed *command line arguments*. C provides a standard argument-passing procedure in which main( ) can accept any number of arguments passed to it. Each argument passed to main( ) is

considered a string and is stored using a pointer array named `argv`. The total number of items on the command line is stored in an integer variable named `argc`.

7. C also provides a `goto` statement. In theory this statement need never be used. In practice it produces confusing and unstructured code. It should be used only in a very limited and controlled manner, if at all.

Appendixes

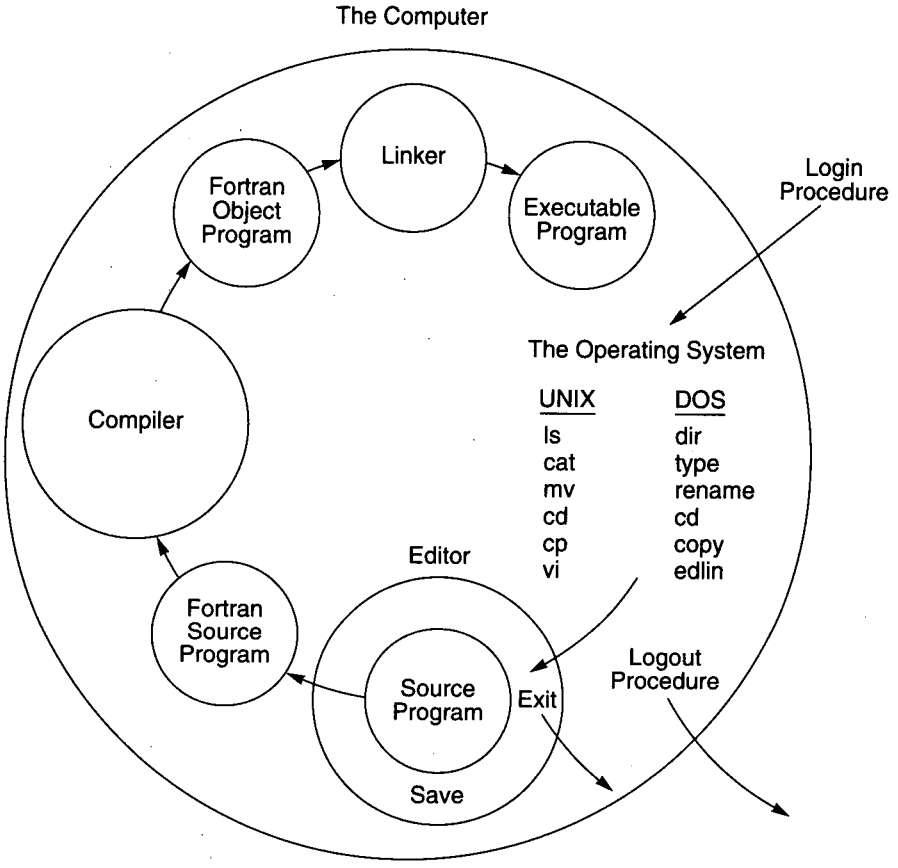
Appendix A Program Entry, Compilation, and Execution under the DOS, UNIX, VAX-VMS, and PRIME Operating Systems

In this appendix, we first examine the steps required to enter, compile, and execute a C program. The specific instructions required by the DOS, UNIX, VAX-VMS, and PRIME operating systems are then provided.

General Introduction

As illustrated in Figure A–1, a computer can be thought of as a self-contained world that is entered by a special set of steps called a login procedure. For some computers such as IBM, Apple, and other desk-top computers, the login procedure is usually as simple as turning the computer's power switch on. Larger multiuser systems, such as DEC, VAX, and PRIME computers, typically require a login procedure consisting of turning a terminal on and supplying an account number and password.

FIGURE A–1 Viewing a Computer as a Self-Contained World



Once you have successfully logged in to your computer system, you are automatically placed under the control of a computer program called the operating system (unless the computer is programmed to switch into a specific application program). The operating system is the program that controls the computer. It is used to access the services provided by the computer, which include the programs needed to enter, compile, and execute a C program.

Communicating with the operating system is always accomplished using a specific set of commands that the operating system recognizes. Although each computer system (IBM, Apple, DEC, PRIME, etc.) has its own set of operating system commands, all operating systems provide commands that allow you to log in to the system, exit from the system, create your own programs, and to quickly list, delete, copy, or rename your programs.

The specific operating system commands and any additional steps used for exiting from a computer, such as turning the power off, are collectively referred to as the logout procedure. Make sure you know the logout procedure for your computer at the time you log in to ensure that you can effectively "escape" when you are ready to leave the system. Since the login and logout procedures for each computer are system dependent, determine these procedures for the system you will be using and list them below:

Login Procedure:_____

_____

Logout Procedure:_____

_____

Specific Operating Systems

Each operating system provides a basic set of commands that allow you to list the names of the programs in the system, type the contents of a program, copy programs, rename programs, and delete programs. Table A–1 lists the operating system commands provided by the IBM DOS, UNIX, VAX-VMS, and PRIME operating systems to perform these and other functions. Space has also been left in the table to list the specific operating system command names used by your system to perform these tasks.

The commands listed in Table A–1 to list, copy, delete, or rename programs are all concerned with manipulating existing programs. Let us now turn our attention to creating, compiling, and executing a new C program. The procedures for doing these tasks are illustrated in Figure A–2. As shown in this figure, the procedure for creating an executable C program consists of three distinct operations: editing (creating or modifying the source code), compiling, and linking. Although every operating system provides an editor program that can be used to create C programs, not all operating systems provide a C compiler. Fortunately, UNIX, VAX, and PRIME operating systems all have a C compiler that is typically installed along with the operating system. For IBM and IBM-compatible PC computers, a separate compiler, such as Borland's Turbo C, Microsoft's Quick C, or Microsoft's standard C compiler must be purchased and installed to provide the capability of compiling C programs.

TABLE A–1 Operating System Commands

Task	DOS	UNIX	VAX	PRIME	Your System
Obtain a directory of programs	dir	ls	dir	LS	
Change to a new directory	cd	cd	cd	DOWN and BACK	
List current directory name	cd	pwd	cd	WHERE	
List a program	type	cat	cat	SLIST	
Copy a program	copy	cp	cp	COPY	
Delete a program	erase	rm	rm	DELETE	
Rename a program	rename	mv	rn	CN	

Editing

Both the creation of a new C program and the modification of an existing C program require the use of an editor program. The function of the editor is to allow a user to type statements at a keyboard and save the typed statements together under a common name, called a source program file name.

As previously illustrated in Figure A–1, an editor program is contained within the environment controlled by the operating system. Like all services provided by the operating system, this means that the editor program can only be accessed using an operating system command. Table A–2 lists operating system commands required by the UNIX, DOS, VAX-VMS, and PRIME operating systems to enter their respective editors. As the UNIX operating system supplies two editor programs, a screen editor named `vi` and a line editor named `ed`, two separate commands are provided in UNIX for accessing the desired editor.

Once the editor program has been requested, the operating system relinquish-

TABLE A–2

Operating System	Command to Enter the Editor	Command to Save and Exit	Command to Exit without Saving
DOS	EDLIN	E	q
UNIX (screen editor)	vi	:wq or ZZ	:q!
UNIX (line editor)	ed	w and then q or ctrl Z	q
VAX-VMS	E	ctrl E	pfi Q
PRIME	PED	.FILE	.QUIT

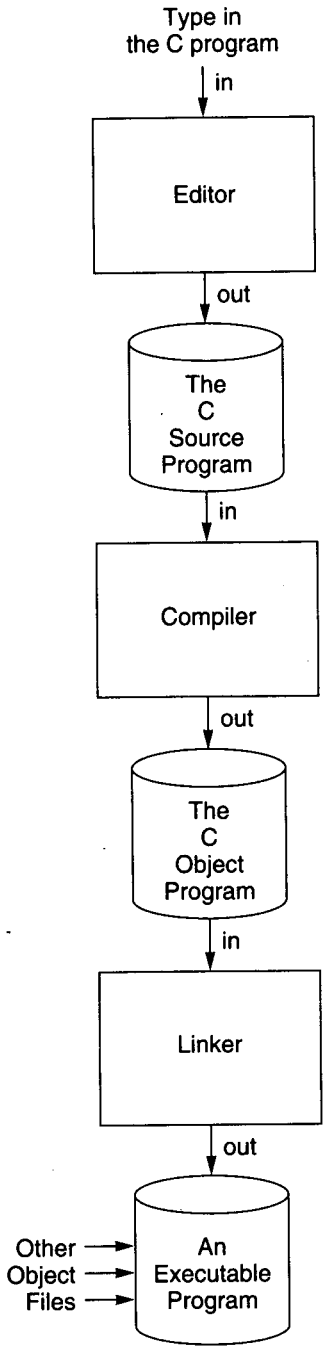


FIGURE A–2 Creating an Executable C Program

es control to this program. Again, as illustrated in Figure A–1, this means that you temporarily leave the world controlled by the operating system and its commands and enter the world controlled by the editor. The editor, like the operating system, has its own set of services and commands. The services provided by the editor include entering C statements, modifying and deleting existing state-

ments in a program, listing a program, naming a program, saving a program, and exiting from the editor back into the operating system with or without saving the program.

In using an editor you must carefully distinguish between entering a C statement and entering an editor command. Some editors make this distinction by using special keys to alert the editor that what is being typed is a command to the editor rather than the line of a program (for example, in BASIC, the line number informs the editor that the entered line is a program statement and the absence of a line number informs the editor that the entered line is an editor command). Other editors, including those listed in Table A–2, contain two modes: a *text mode* for entering and modifying program statements, and a *command mode* for entering editor commands. Table A–3 lists the commands provided by the UNIX, DOS, VAX-VMS, and PRIME editors for alerting the editor as to whether the text being typed is a command or a program statement.

In command mode, each editor permits you to perform the tasks listed in Table A–4. Once you have determined the editor you will be using, fill in Table A–4 (for some of these tasks, the required commands can be found in Tables A–2 and A–3).

Compiling and Linking

Translating a C source program into a form that can be executed by the computer is accomplished using a *compiler* program. The output produced by the compiler is called an *object* program. An object program is simply a translated version of the source program that can be executed by the computer system with one more processing step. Let us see why this is so.

Most C programs contain statements that use preprogrammed routines, called intrinsic functions, for finding such quantities as square roots, logarithms, trigonometric values, absolute values, or other commonly encountered mathematical calculations. Additionally, a large C program may be stored in two or more separate program files. However, multiple files must ultimately be combined to form a single program before the program can be executed. In both of these cases it is the task of the *linker* to combine all of the intrinsic functions and individual object files into a single program ready for execution. This final program is called an executable program.

TABLE A–3 Switching between Command and Text Modes

Editor	Commands to Enter Text Mode from Command Mode	Commands to Enter Command Mode from Text Mode
DOS—EDLIN	i or type at line no.	ctrl and C keys
UNIX—vi	a, i, o, c, s	Esc key
UNIX—ed	a, i, o, c, s	.(period)
VAX—EDT	c	ctrl Z
PRIME—PED	always in text mode	.followed by command

TABLE A–4 Editor Commands (fill in with your editor's commands)

Task	Command	Example
Save the program and exit from the editor		
Save the program without exiting from the editor		
Exit from the editor without saving the program		
Switch to text mode (if applicable)		
Switch to command mode (if applicable)		
List the complete program from within the editor		
List a set of lines from within the editor		
List a single line from within the editor		
Delete the complete program from within the editor		
Delete a set of lines from within the editor		
Delete a single line from within the editor		
Name a program from within the editor		

Both the compiler and the linker programs can be assessed using individual operating system commands. For ease of operation, however, all operating systems that provide a C compiler also provide a single command that both compiles a C program and links it correctly with any required other object programs using one command. Table A–5 lists the commands required by the UNIX, VAX-VMS, and PRIME operating systems to either compile only, link only, or compile and link a C program to produce an executable program. (Since DOS does not provide a C compiler, no entry is included in Table A–5 for this operating system.) Space has been left in the table to enter the command used by your computer for performing these operations.

Finally, once the C source program has been compiled and linked, it must be run. In both the VAX-VMS and PRIME operating systems execution of the executable program is begun by simply typing the name of the program in response to the operating system prompt. In the UNIX operating system the executable program produced by the linker is named `a.out`. Thus, for the UNIX operating system the execution of the last compiled and linked C program is initiated by typing `a.out` in response to the oeprating system prompt. Determine and then list the command used by your computer for performing this operation:

Operating system command to
execute a compiled and linked program:_____

TABLE A–5 Specific Operating System Compile and Link Commands

Operating System	Compile and Link Command	Compile Only Command	Link Command
UNIX	cc filename(s)	cc filename(s) -c	ld objectname(s) -ic
VAX-VMS	—	cc filename	lin filename
PRIME	clg c -br l	cc filename	bind :li ccmain :lo filename :li g_lib :li
Your System			

Note: For each operating system listed in Table A–5. every source filename being compiled must end in a .c, and every object filename being linked must end in a .o. The output of a compile only command automatically produces an equivalent .o object file if the compilation is successful.

Appendix B Using Borland's Turbo C Compiler

Borland's Turbo C compiler provides a complete integrated programming development system that permits the user to enter, edit, compile, link, and execute C programs. Additionally, it provides a conventional command-line version. In this appendix both versions of the compiler are described. Before reading this appendix, however, you should understand the introductory material on program entry, compilation, and execution presented in the general introduction to Appendix A.

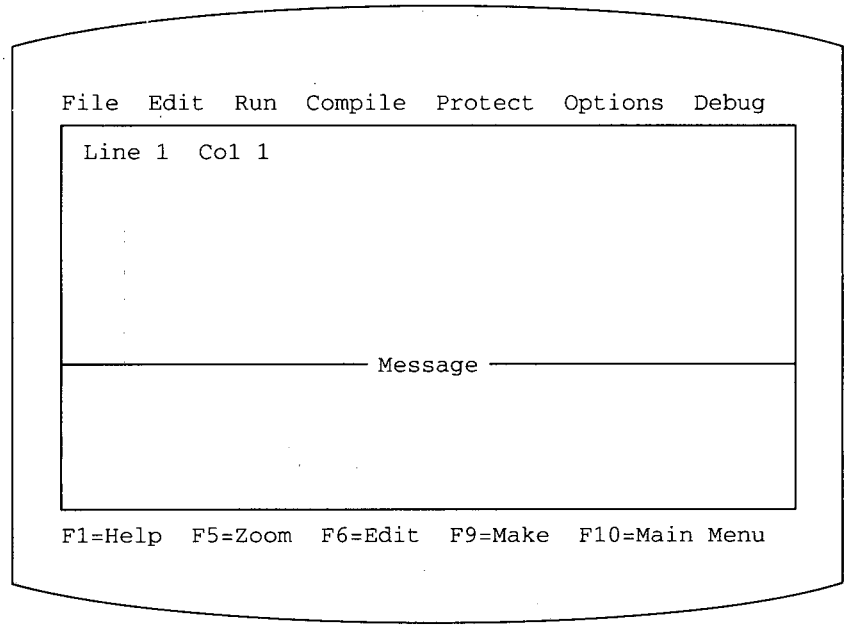
The integrated development environment of the Turbo C compiler is a menu-driven program, which is invoked by typing `TC` in response to the system prompt. Assuming the system prompt is `C>`, the command

```
C> TC
```

invokes the integrated development version of the Turbo C compiler. This version permits the programmer to create, edit, compile, link, and run a program using a choice of menu options provided by Turbo C. In response to the `TC` command, the Turbo C start-up window illustrated in Figure B–1 is displayed.

The choices available to the user are listed in the menu bar at the top of the window. Once this start-up window is displayed, you can select `File`, `Edit`, `Run`, `Compile`, `Protect`, `Options`, or `Debug`. The selection is made by either pressing the `Alt` key and the first letter of the desired choice (for example, pressing the `Alt` key and `F` key simultaneously selects the `File` choice), or by moving to the desired choice using the cursor keys and then pressing the `Enter` key.

FIGURE B–1



To initially enter a C program, you must select the `File` option. Once this option is selected, the `File` window illustrated in Figure B–2 is displayed. From this `File` window select the `Load` option by typing the letter `L`. A box is then presented that allows you to type in the name of your C program. In this box enter the name you wish to give your program, such as `hello.c`, and push the `Enter` key. (Although Turbo C does not require that your C program's name end in a period followed by the `c`, this is a convention that is required by most other compilers and used by most Turbo C programmers.)

After you have named youir program and pushed the `Enter` key, Turbo C transfers control to its editor. In editor mode, you can type your program and move to any desired area by using the cursor arrow keys. Most of the editor commands are the same as those used in the WordStar word processing program. Of these commands, the ones that are used most often are listed in Table B–1.

After you have completed typing and editing your program, it can be compiled, linked, and run by pushing the `Alt` and `R` keys at the same time. This combination selects the `Run` option from the start-up window. At this point, Turbo C attempts to compile, link, and begin execution of your program. If your program has no errors an executable version of your program is created and automatically run. For example, if our `hello.c` program successfully compiles and links, the executable program `hello.exe` is created and automatically saved before it is run by Turbo C.

Should an error occur in either the compilation or linking stages, the program returns control to the editor, an error message is displayed in a message window at the bottom of the screen, and the cursor is automatically placed on the line in the program at which the error is detected. Unfortunately, this sometimes can be misleading as the error may actually be on the previous line. For example, the C code

```
int a
b = 22;
```

will indicate an error on the line `b = 22;`. The actual error occurs, however, because the line immediately preceding this is not terminated by a semicolon. Since white space is ignored by all C compilers (see Section 1.4), these two lines are read as

```
int a b = 22;
```

FIGURE B–2 The Turbo C File Window

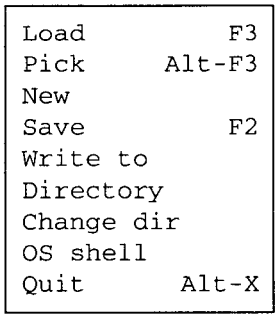


TABLE B–1 Commonly Used Editing Commands

Command	Description
Ctrl and y keys pushed simultaneously	Delete the line of text containing the cursor
Ctrl and t keys pushed simultaneously	Delete the word containing the cursor
Ctrl and k keys pushed simultaneously, followed by pushing the B key	Mark the start of a block starting at the cursor
Ctrl and k keys pushed simultaneously, followed by pushing the k key	Mark the end of a block immediately to the left of the cursor
Ctrl and k keys pushed simultaneously, followed by pushing the C key	Put a copy of the marked block immediately to the right of the current cursor position
Ctrl and k keys pushed simultaneously, followed by pushing the V key	Move the marked block immediately to the right of the current cursor position

Once the semicolon is detected, the C compiler realizes an error has occurred and locates the cursor on what it detects as the offending line, namely, `b = 22;`.

To correct the error, you must push the `F6` key to move control from the message window into the program and then make your corrections. If multiple errors are indicated, pushing the `F6` key once more places you back in the message window at the next error message. This process of moving from the message window into the program to make the corrections is continued until all of your corrections have been made. At this point another run can be attempted by pushing the `Alt` and `R` keys simultaneously.

At any time during the editing process you can save a copy of your program by pushing the `Alt` and `F` keys simultaneously and then pushing the `S` (for `Save`) key. To quit the Turbo C program and return to the operating system, you may either push the `Alt` and `X` keys simultaneously, or push the `Alt` and `F` keys to get back into the `File` option and then push the `Q` key (for quit).

Once you are back at the system prompt you may run any C program that was successfully compiled and linked by Turbo C. For example, and again assuming that the system prompt is `C>`, typing

```
C> hello
```

would cause the `hello` program to begin execution.

In addition to the integrated environment provided by Turbo C, a command-line version is also available. In the command line version an editor must first be used to create a C program. Once the C program has been created, it can be compiled and linked using the command `TCC`. For example, issuing the command

```
C> TCC hello.c
```

would cause the compiler to compile and link the `hello.c` program. If the compilation and linkage is successful, the executable version of this program, named

`hello.exe`, is created. The executable version can be run by issuing the command

```
C> hello
```

If the compilation is unsuccessful, the editor must be invoked and the errors corrected before recompilation.

Appendix C Using Microsoft's Quick C Compiler

Microsoft's *Quick C* compiler provides a complete integrated programming development system that lets the user enter, edit, compile, link, and execute C programs. Additionally, it provides a conventional command-line version. In this appendix, both versions of the compiler are described. Before reading this appendix, however, you should understand the introductory material on program entry, compilation, and execution presented in the general introduction to Appendix A.

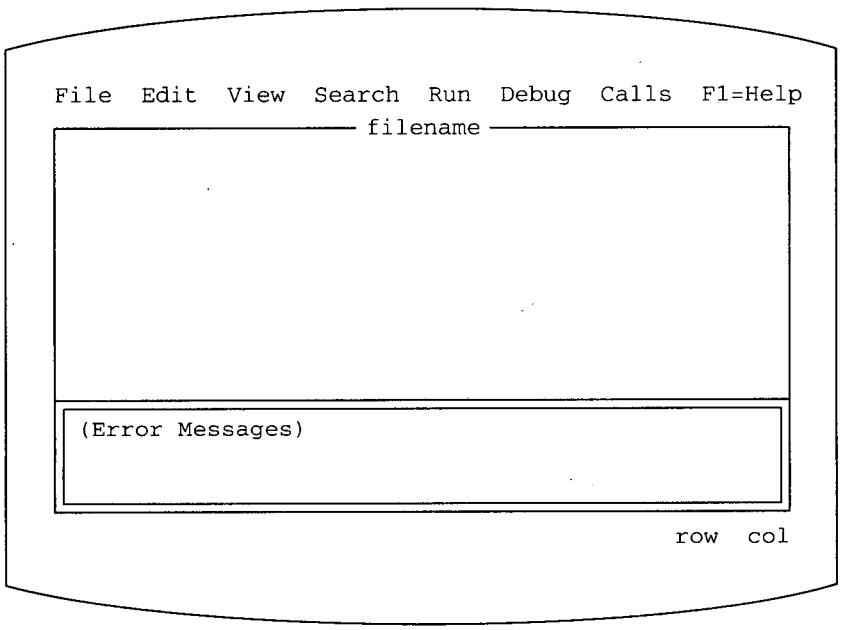
The Microsoft Quick C compiler is a menu-driven program, which is invoked by typing `QC` in response to the system prompt. Assuming the system prompt is `C>`, the command

```
C> QC
```

invokes the integrated development version of the Quick C compiler. This version permits the programmer to create, edit, compile, link, and run a program using a choice of menu options provided by Quick C. In response to the `QC` command the Quick C start-up window illustrated in Figure C–1 is displayed.

The choices available to the user are listed in the menu bar at the top of the window. Once this start-up window is displayed, you have a choice of selecting `File`, `Edit`, `View`, `Search`, `Run`, `Debug`, or `Calls`. The selection is made by either pressing the `Alt` key and the first letter of the desired choice (for example, pressing the `Alt` key and `F` key at the same time selects the File choice), or

FIGURE C–1 The Quick C Start-Up Window



by moving to the desired choice using the cursor keys and then pressing the Enter key. Table C–1 lists the options provided by each menu choice.

To initially enter a C program, you must select the File option. Once this option is selected, the File menu illustrated in Figure C–2 is displayed. From this File menu select the New option by typing the letter N if you will be creating a new program or select the Open option by typing the letter O if you want to load an existing file. When you select the O option a box is presented that allows you to select the name of your existing C program.

After you have selected either a new or existing program, Quick C transfers control to its editor. In editor mode you can type in your program and move to any desired point in the program using the cursor arrow keys. Table C–2 lists the most commonly used editor commands, many of which are the same as those used in the WordStar word processing program.

After you have completed typing and editing your program, it can be compiled, linked, and run by pushing the Alt and R keys at the same time. This combination selects the Run menu, from which the Start option must by selected. Alternatively, holding the Shift and F5 keys down together while in the editing mode also initiates the compilation and execution of the program being edited. If your program has no errors, an executable version of your program is created and automatically run. For example, if our hello.c program successfully compiles and links, the executable program hello.exe would be created and automatically saved before it is run by Quick C.

Should an error occur in either the compilation or linking stages, the program returns control to the editor, and error message is displayed in a message window at the bottom of the screen, and the cursor is automatically placed on the line in the program at which the error is detected. Unfortunately, this sometimes can be misleading as the error may actually be on the previous line. For example, the C code

```
int a
b = 22;
```

TABLE C–1 Quick C Menu Options

Menu	Options provided
File	Create, load, merge, print, and save source files. Also provides access to the operating system and exit from Quick C.
Edit	Add, modify, copy, and delete source text.
View	Customize the display of programs and alter the programming environment.
Search	Find, replace, and display source file text.
Run	Compile and run C programs.
Debug	Select and control debugging features.
Calls	Display the hierarchy of function calls specified by a C program.

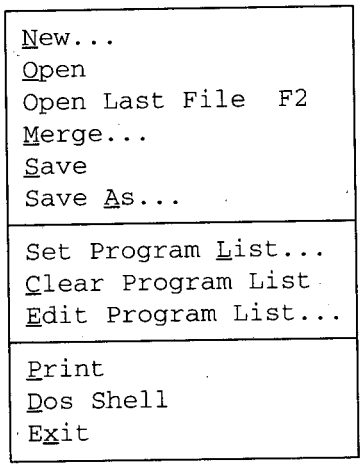


FIGURE C–2 The Quick C file Menu

will indicate an error on the line `b = 22;`. The actual error occurs, however, because the line immediately preceding this is not terminated by a semicolon. Since white space is ignored by all C compilers (see Section 1.4), these two lines are read as

```
int a b = 22;
```

Once the semicolon is detected, the compiler realizes an error has occurred and locates the cursor on what it detects as the offending line, namely, `b = 22;`.

If multiple errors are indicated, pressing the `Shift` and `F3` keys simultaneously moves the cursor to the line in the program that caused the next error and displays the error message in the error window. Similarly, pressing the `Shift` and `F4` keys simultaneously moves the cursor to the line in the program that caused the previous error and displays the previous error message in the error window. When all errors have been corrected another run can be attempted by pushing the `Shift` and `F5` keys simultaneously.

At any time during the editing process, you can save a copy of your program by pushing the `Alt` and `F` keys simultaneously and then pushing the `S` (for save)

TABLE C–2 Commonly Used Editor Commands

Command	Description
Ctrl and y keys pushed simultaneously	Delete the line of text containing the cursor
Ctrl and t keys pushed simultaneously	Delete the word containing the cursor
Shift and cursor key	Mark a block of characters
Ctrl and Ins keys pushed simultaneously	Put a copy of the marked block immediately to the left of the current cursor position

key. To quit the program and return to the operating system you may either push the Alt and X keys simultaneously, or push the Alt and F keys to get back into the File option and then push the X key.

Once you are back at the system prompt, you may rerun any C program that was successfully compiled and linked by Quick C. For example, and again assuming that the system prompt is C>, typing

```
C> hello
```

would cause the hello program to begin execution.

In addition to the integrated environment provided by Quick C, a command-line version is also available. In the command-line version and editor must first be used to create a C program. Once the C program has been created, it can be compiled and linked using the command QCL. For example, issuing the command

```
C> QCL hello.c
```

would cause the compiler to compile and link the hello.c program. If the compilation and linkage is successful, the executable version of this program, named hello.exe, is created. The executable version can then be run by issuing the command

```
C> hello
```

If the compilation is unsuccessful, the editor must be invoked and the errors corrected before recompilation.

Appendix D Living with an Old C Compiler*

The reference manual contained in Kernighan and Ritchie's 1978 classic book *The C Programming Language* provided the original definition of C. While ANSI C preserves the spirit of the original C, it has incorporated various enhancements to the original C and modern developments in programming languages. Most of the changes, however, are minor. Considering that some of you may still have pre-ANSI-C compilers, we discuss important differences between the original and ANSI C, so that you may be able to write C programs for these compilers as well.

Most Notable Change

ANSI C differs from the original C most notably in the way a function is defined and used.

Function Definition

The general form for defining a function in the original C is as follows:

```
function-type   function-name (argument-list)
   argument-declaration
{
   variable-declarations

   function-statements
}
```

Only the argument names are specified in the *argument-list*. Their types are specified in the *argument-declarations*, following the argument list. An argument whose type is not specified is taken to be of type `int`. If a function does not have any arguments, it is indicated by an empty pair of parentheses ( ), following the function name. The specification of function-type is optional, if it is `int`.

Here are some examples of function definitions in ANSI C and their corresponding definitions in the original C:

ANSI C

```
int main (void)
{
  ...
}

int quotient (int i, int j)
{
  ...
}

double chebyshev(int n, float x)
{
  ...
```

Original C

```
main ( )
{
  ...
}

int quotient (i, j)
{
  ...
}

double chebyshev(n, x)
  int n;
  float x;
```

*Reprinted from *Programming in ANSI C* by Ram Kumar and Rakesh Agrawal (Saint Paul: West Publishing Co., 1992), 425-32. Reprinted with permission.

ANSI C

```
}

char *cmp (char s[], int n, char *t)
{
  ...
}

void mktree (struct info **parent)
{
  ...
}
```

Original C

```
{
  ...
}

char *cmp (s, n, t)
  char s [], *t;
  int n;
{
  ...
}

mktree (parent)
  struct info **parent;
{
  ...
}
```

In the first example, the return type of `main` has not been specified in the original C version, as it is taken to be `int` by default. Instead of `void`, the empty pair of parentheses following the function name are used to specify that `main` takes no arguments. In the `quotient` example, the types of the arguments `i` and `j` have not been specified, and they are taken to be of type `int`. But remember that the omission of type specification is a bad programming practice, and you should avoid it. You should also specify the return type of a function, even if it is `int`, as we have done for `quotient`. The `chebyshev` example shows the type declarations for arguments. Note that each argument declaration is terminated with a semicolon. The `cmp` example shows that arguments of the same type can be declared together in one declaration, and that the arguments need not be Here declared in the same order in which they appear in the argument list. Finally, the `mktree` example illustrates the convention of not specifying the return type for a function that does not return a value, if the type `void` is not supported by the compiler.

Function Call

In the original C, no declaration is required in the calling function for a function being called that returns an `int`. However, if a called function returns a value other than `int` and its declaration is not available, it must be explicitly declared in the calling function using a declaration of the form

function-type function-name( );

Unlike ANSI C function prototypes, argument names and their types are not specified in the declaration. The following example shows the declaration and call to the `cmp` function in ANSI C and the original C:

ANSI C

```
char *scmp (char s[], char *t, int n)
{
  char *cmp(char [], int, char *);
  int safestr(char [], int);

  return safestr(s,n) ?
         cmp(s,n,t): NULL;
}
```

Original C

```
char *scmp (s, t, n)
  char s [], *t;
  int n;
{
  char *cmp();

  return safestr(s,n) ?
       cmp(s,n,t): NULL;
}
```

In the original C version, it has been specified that `cmp` returns a `char *`, but the types of its arguments have not been specified. Also, no declaration has been given for `safestr`, as it returns an `int`.

Argument Binding

The type conversion rules at the time of binding actual arguments to the formal arguments are different in ANSI and the original C. In the original C, when an expression appears as an argument in a function call, adjustments are made in the type of the value of the expression, using the unary conversion rules. In particular, arguments of type `float` are converted to `double`, and of type `char` or `short` to `int`, before being passed as arguments. Adjustments are also made to the types of the function's formal arguments, and parameters of type `char`, `short`, or `float` are implicity promoted to `int`, `int`, and `double` respectively. No other conversions are performed automatically; appropriate casts must be used to effect necessary conversions. The following example illustrates this difference:

ANSI C

```
double poly(int i, int j)
{
  double chebyshev(int, float);

  return chebyshev(i,j);
}
```

Original C

```
double poly(i, j)
  int i, j;
{
  double chebyshev();

  return chebyshev(i, (float) j);
}
```

In ANSI C, when a function for which a prototype has been specified is called, the actual arguments to the function are converted, as if by assignment, to the declared types of the formal arguments. Thus, the call

```
chebyshev (i, j);
```

is equivalent to

```
chebyshev (i, (float) j):
```

and no explicit casting is necessary.

Since the original C does not have function prototypes, the actual arguments are not checked for consistency with the formal arguments. Functions can potentially take an arbitrary number of arguments of arbitrary types. Thus, an original C compiler does not detect an error in the following call to `cmp`:

```
return safestr (s,n) ? cmp (s,t,n) : NULL;
```

in which `t`, a `char*`, is being passed to an `int`, to a `char*`. It also does not detect an error in the following call to `safestr`:

```
return safestr (s,t,n) ? cmp (s,t,n) : NULL;
```

in which there is an extra argument and the types of the second actual argument

and formal argument do not match. Similarly, the error that `safestr` has been provided one less argument in the following call:

```
return safestr(s) ? cmp(s,t,n) : NULL;
```

is also not detected. Mismatch in number and types of actual arguments and formal arguments is the most common error found in programs written in the original C, and you should always ensure that the caller provides the correct number of arguments of consistent types, and that the callee does not use an unsupplied argument.

Work-Arounds

ANSI C provides some convenient features not available in the original C. However, you may get by in many instances by using alternative facilities.

Size of an Identifier

ANSI C permits a minimum of 31 significant characters in identifiers, whereas the original C specifies that only the first eight characters of an identifier are significant. Thus, variable names like `average_weight` and `average_width` may not be distinguishable in the original C, as they are identical up to the first eight characters. Use small variable names.

String Concatenation

The original C does not concatenate adjacent string constants. Write them as one constant and use a backslash at the end of a line to continue a long string onto the next line.

Automatic Type Conversion

The automatic type conversion rules in the original C are somewhat different. Keep in mind the following figure when writing expressions involving mixed types:

Predefined Data Types

The types `size_t` and `ptrdiff_t` are not standard in the original C. Use `int`, instead.

Generic Pointer

ANSI C uses the type `void*` as the proper type for a *generic* pointer. The type `char*` plays this role in the original C.

`switch` *Statement*

The `switch` statement control and case label expressions may only be of type `int` (not any integral type) in the original C. Use the `if` statement for other integral types.

Initialization of Automatic Arrays and Structures

ANSI C permits the initialization of automatic arrays and structures, a feature not permitted in the original C. You should replace initialization with explicit assignments in such cases, as shown in the following example:

ANSI C

```
void foo(void)
{
  int a[2] = {0, 1};
  struct pt
    {
      float x, y;
    } p = {0, 0};
  ...

}
```

Original C

```
foo()
{
  int a[2];
  struct pt
    {
      float x, y;
    } p;
  a[0] = 0, a[1] = 1;
  p.x = p.y = 0;
  ...
}
```

However, the original C does allow initialization of static arrays and structures. Therefore, if you do not need to subsequently change the elements of the arrays or structures to be initialized, you may declare them to be `static` and initialize them, as shown in the following example:

ANSI C

```
void foo(void)
{
  int a[2] = {0, 1};
  struct pt
    {
      float x, y;
    } p = {0, 0};
  ...
}
```

Original C

```
foo()
{
  static int a[2] = {0, 1};
  static struct pt
    {
      float x, y;
    } p = {0, 0};
  ...
}
```

Initialization of Unions

ANSI C allows restricted initialization of union variables, whereas the original C does not allow any union variable (including `static`) to be initialized. Replace initializations with explicit assignments.

Structures and Unions in Assignments and Function Definitions

In ANSI C, a structure variable may be assigned to another structure variable, structures may be passed as function arguments, and functions may return structures. These features are not available in the original C. However, you may replace structure assignment by assignments to individual members; instead of passing a structure as argument, you may either pass structure members or a pointer to the structure; and a function may return a pointer to the structure, rather than returning the structure. The following example illustrates these workarounds:

ANSI C

```
struct pt
{
float x, y;
} p = {0, 0};

void bar(void)
{
struct pt q, r;
float dist(struct pt), f;
struct pt polar(struct pt);

q = p
f = dist(q);
r = polar(q);
...
}  }
```

Original C

```
struct pt
{
  float x, y;
} p = {0, 0};

bar()
{
  struct pt q, *r;
  float dist(/* int, int */);
struct pt *polar(/* struct pt * */);

  q.x = p.x, q.y = p.y;
  f = dist(q.x, q.y);
  r = polar(&q);
```

The preceding observations also hold for union types.

Enumeration Types

The original C does not provide enumeration types. However, their functionality can be partially simulated using `#defines` and `typedefs`, as shown in the following example:

ANSI C

```
typedef enum {chablis,claret} wine;

...
wine bordeaux, *burgundy;
...
bordeaux = claret;
*burgundy = chablis;
```

Original C

```
#define chablis 0
#define claret  1
typedef int wine;
...
  wine bordeau, *burgundy;
...
bordeaux = claret;
*burgundy = chablis;
```

Standard Library

The "standard" library is not standard in the original C, and the functions present in one implementation may differ from those in another. However, you should be able to find most of the functions defined in the standard library by ANSI C in your implementation's library, although sometimes under different names.

Irreconcilables

The following are some of the important ANSI C features for which there are no analogs in the original C:

- A standard and portable facility for writing frunctions that take a variable number of arguments.
- The type qualifiers `const` and `volatile` to indicate special properties of the objects being declared.

- The extra-precision type `long double`.
- The facility to explicitly specify the signedness of characters and other types by using the keywords `signed` and `unsigned`.
- The suffixes `U` or `L` for integers and `F` or `L` for reals, to make the types of constants explicit.
- The notation for expressing a hexadecimal constant by a bit pattern of the form '`\x`*h*'.
- Trigraph sequences as alternative spellings of C characters not in the ASCII invariant code set, and the data type `wchar_t` for wide characters.
- The preprocessing operators `defined`, #, and # #.
- Flexibility to allow whitespaces to precede or follow the directive-introducing character #.
- The preprocessing directives `#elif` and `#pragma`.
- The `#include` directive of the form

  `#include` *token-sequence*

- The special macro names `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, and `__STDC__`.

Appendix E Operator Precedence Table

Table E–1 presents the symbols, precedence, descriptions, and associativity of C's operators. Operators toward the top of the table have a higher precedence than those toward the bottom. Operators within each box have the same precedence and associativity.

TABLE E–1 Summary of C Operators

Operator	Description	Associativity
`()`	Function call	Left to right
`[]`	Array element	
`->`	Structure member pointer reference	
`.`	Structure member reference	
`++`	Increment	Right to left
`--`	Decrement	
`-`	Unary minus	
`!`	Logical negation	
`~`	One's complement	
`(type)`	Type conversion (cast)	
`size of`	Storage size	
`&`	Address of	
`*`	Indirection	
`*`	Multiplication	Left to right
`/`	Division	
`%`	Modulus (remainder)	
`+`	Addition	Left to right
`-`	Subtraction	
`<<`	Left shift	Left to right
`>>`	Right shift	
`<`	Less than	Left to right
`<=`	Less than or equal to	
`>`	Greater than	
`>=`	Greater than or equal to	
`==`	Equal to	Left to right
`!=`	Not equal to	
`&`	Bitwise AND	Left to right
`^`	Bitwise exclusive OR	Left to right
`|`	Bitwise inclusive OR	Left to right
`&&`	Logical AND	Left to right
`||`	Logical OR	Left to right
`?:`	Conditional expression	Right to left

TABLE E–1 Summary of C Operators (Continued)

Operator	Description	Associativity
=	Assignment	Right to left
+= -= *=	Assignment	
/= %= &=	Assignment	
^= |=	Assignment	
<<= >>=	Assignment	
,	Comma	Left to right

Appendix F ASCII Character Codes

Key(s)	Dec	Oct	Hex	Key	Dec	Oct	Hex	Key	Dec	Oct	Hex
Ctrl 1	0	0	0	+	43	53	2B	V	86	126	56
Ctrl a	1	1	1	,	44	54	2C	W	87	127	57
Ctrl b	2	2	2	-	45	55	2D	X	88	130	58
Ctrl c	3	3	3	.	46	56	2E	Y	89	131	59
Ctrl d	4	4	4	/	47	57	2F	Z	90	132	5A
Ctrl e	5	5	5	0	48	60	30	[	91	133	5B
Ctrl f	6	6	6	1	49	61	31	\	92	134	5C
Ctrl g	7	7	7	2	50	62	32	]	93	135	5D
Ctrl h	8	10	8	3	51	63	33	^	94	136	5E
Ctrl i	9	11	9	4	52	64	34	_	95	137	5F
Ctrl j	10	12	A	5	53	65	35	`	96	140	60
Ctrl k	11	13	B	6	54	66	36	a	97	141	61
Ctrl l	12	14	C	7	55	67	37	b	98	142	62
Ctrl m	13	15	D	8	56	70	38	c	99	143	63
Ctrl n	14	16	E	9	57	71	39	d	100	144	64
Ctrl o	15	17	F	:	58	72	3A	e	101	145	65
Ctrl p	16	20	10	;	59	73	3B	f	102	146	66
Ctrl q	17	21	11	<	60	74	3C	g	103	147	67
Ctrl r	18	22	12	=	61	75	3D	h	104	150	68
Ctrl s	19	23	13	>	62	76	3E	i	105	151	69
Ctrl t	20	24	14	?	63	77	3F	j	106	152	6A
Ctrl u	21	25	15	@	64	100	40	k	107	153	6B
Ctrl v	22	26	16	A	65	101	41	l	108	154	6C
Ctrl w	23	27	17	B	66	102	42	m	109	155	6D
Ctrl x	24	30	18	C	67	103	43	n	110	156	6E
Ctrl y	25	31	19	D	68	104	44	o	111	157	6F
Ctrl z	26	32	1A	E	69	105	45	p	112	160	70
Esc	27	33	1B	F	70	106	46	q	113	161	71
Ctrl <	28	34	1C	G	71	107	47	r	114	162	72
Ctrl /	29	35	1D	H	72	110	48	s	115	163	73
Ctrl =	30	36	1E	I	73	111	49	t	116	164	74
Ctrl -	31	37	1F	J	74	112	4A	u	117	165	75
Space	32	40	20	K	75	113	4B	v	118	166	76
!	33	41	21	L	76	114	4C	w	119	167	77
"	34	42	22	M	77	115	4D	x	120	170	78
#	35	43	23	N	78	116	4E	y	121	171	79
$	36	44	24	O	79	117	4F	z	122	172	7A
%	37	45	25	P	80	120	50	{	123	173	7B
&	38	46	26	Q	81	121	51	|	124	174	7C
'	39	47	27	R	82	122	52	}	125	175	7D
(	40	50	28	S	83	123	53	~	126	176	7E
)	41	51	29	T	84	124	54	del	127	177	7F
*	42	52	2A	U	85	125	55				

Appendix G Floating Point Number Storage

The two's complement binary code used to store integer values was presented in Section 1.7. In this appendix we present the binary storage format typically used in C to store single precision and double precision numbers, which are stored as floats and doubles, respectively. Collectively, both single and double precision values are commonly referred to as floating point values.

Like their decimal number counterparts that use a decimal point to separate the integer and fractional parts of a number, floating point numbers are represented in a conventional binary format with a binary point. For example, consider the binary number 1011.11. The digits to the left of the binary point (1011) represent the integer part of the number and the digits to the right of the binary point (11) represent the fractional part.

To store a floating point binary number a code similar to decimal scientific notation is used. To obtain this code the conventional binary number format is separated into a mantissa and an exponent. The following examples illustrate floating point numbers expressed in this scientific notation.

Conventional Binary Notation	Binary Scientific Notation
1010.0	1.01 exp 011
−10001.0	-1.0001 exp 100
0.001101	1.101 exp -011
−0.000101	-1.01 exp -100

In binary scientific notation, the term exp stands for exponent. The binary number in front of the exp term is the mantissa and the binary number following the exp term is the exponent value. Except for the number zero, the mantissa always has a single leading 1 followed immediately by a binary point. The exponent represents a power of 2 and indicates the number of places the binary point should be moved in the mantissa to obtain the conventional binary notation. If the exponent is positive, the binary point is moved to the right. If the exponent is negative, the binary point is moved to the left. For example, the exponent 011 in the number

1.01 exp 011

means move the binary point three places to the right, so that the number becomes 1010. The -011 exponent in the number

1.101 exp -011

means move the binary point three places to the left, so that the number becomes

.001101

In storing floating point numbers, the sign, mantissa, and exponent are stored

TABLE G–1 IEEE Standard 754-1985 Floating Point Specification

Data Format	Sign Bits	Mantissa Bits	Exponent Bits
Single Precision	1	23	8
Double Precision	1	52	11
Extended Precision	1	64	15

individually within separate fields. The number of bits used for each field determines the precision of the number. Single precision (32 bit), double precision (64 bit), and extended precision (80 bit) floating point data formats are defined by the Institute of Electrical and Electronics Engineers (IEEE) Standard 754-1985 to have the characteristics given in Table G–1. The format for a single precision floating point number is illustrated in Figure G–1.

The sign bit shown in Figure G–1 refers to the sign of the mantissa. A sign bit of 1 represents a negative number and a zero sign bit represents a positive value. Since all mantissas, except for the number zero, have a leading 1 followed by their binary points, these two items are never stored explicitly. The binary point implicitly resides immediately to the left of mantissa bit 22, and a leading 1 is always assumed. The binary number zero is specified by setting all mantissa and exponent bits to 0. For this case only, the implied leading mantissa bit is also zero.

The exponent field contains an exponent that is biased by 127. For example, an exponent of 5 would be stored using the binary equivalent of the number 132 (127 + 5). Using eight exponent bits, this is coded as 100000100. The addition of 127 to each exponent allows negative exponents to be coded within the exponent field without the need for an explicit sign bit. For example, the exponent -011, which correspond to -3, would be stored using the binary equivalent of +124 (127 - 3).

Figure G–2 illustrates the encoding and storage of the decimal number 59.75 as a 64-bit single precision binary number. The sign, exponent, and mantissa are determined as follows. The conventional binary equivalent of

$$-59.75$$

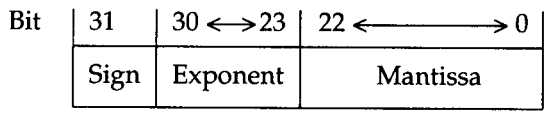
is

$$-111011.11$$

Expressed in binary scientific notation this becomes

$$-1.1101111 \text{ exp } 101$$

FIGURE G–1 Single Precision Floating Point Number Storage Format



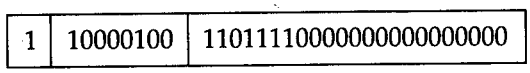


FIGURE G–2 The Encoding and Storage of the Decimal Number 59.75

The minus sign is signified by setting the sign bit to 1. The mantissa's leading 1 and binary point are omitted and the 23-bit mantissa field is encoded as

11011110000000000000000

The exponent field encoding is obtained by adding the exponent value of 101 to 1111111, which is the binary equivalent of the 127_{10} bias value:

$$\begin{array}{rcl} 1\ 1\ 1\ 1\ 1\ 1\ 1 & = & 127_{10} \\ +\ 1\ 0\ 1 & = & +5_{10} \\ \hline 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0 & = & 132_{10} \end{array}$$

Appendix H Solutions

Section 1.1

1. a. A computer program is a sequence of instructions used to operate a computer to produce a specific result. (A more formal definition is that a program is the description of an algorithm using a computer language.)
b. Programming is the process of writing instructions in a language that a computer can respond to and that other programmers can understand.
c. A programming language is a set of instructions that can be used to construct a computer program.
d. An algorithm is a step-by-step sequence of instructions that describes how a computation or task is to be performed.
e. pseudocode is a method of describing an algorithm using English-like phrases.
f. A flowchart is a description of an algorithm using specifically defined graphical symbols for input, output, and processing operations that use flow lines to connect each graphical symbol.
g. A source program is written in a computer language, such as C, and must be translated into a computer's machine language before it can be executed.
h. An object program is the machine language (translated) version of a source program.
i. A compiler is a program that translates a source program into an object program.
j. An interpreter executes a source program by translating and immediately executing each instruction in the source program as it is encountered. Using an interpreter, no object program is produced.

3. Step 1: Pour the contents of the first cup into the third cup
Step 2: Rinse out the first cup
Step 3: Pour the contents of the second cup into the first cup
Step 4: Rinse out the second cup
Step 5: Pour the contents of the third cup into the second cup

5. Step 1: Compare the first number with the second number and use the smallest of these numbers for the next step
Step 2: Compare the smallest number found in step 1 with the third number. The smallest of these two numbers is the smallest of all three numbers.

7. a. Step 1: Compare the first name in the list with the name JONES. If the names match, stop the search; else go to step 2.
Step 2: Compare the next name in the list with the name JONES. If the names match, stop the search; else repeat this step.

Section 1.2

1.

`m1234( )`	Valid. Not a mnemonic.
`new_bal( )`	Valid. A mnemonic.
`abcd( )`	Valid. Not a mnemonic.
`A12345( )`	Valid. Not a mnemonic.
`1A2345( )`	Invalid. Violates Rule 1; starts with a number.
`power( )`	Valid. A mnemonic.
`abs_val( )`	Valid. A mnemonic.
`invoices( )`	Valid. A mnemonic.
`do( )`	Invalid. Violates Rule 3; is a reserved word.
`while( )`	Invalid. Violates Rule 3; is a reserved word.
`add_5( )`	Valid. Could be a mnemonic.
`taxes( )`	Valid. A mnemonic.
`net_pay( )`	Valid. A mnemonic.

`12345( )`	Invalid. Violates Rule 1; starts with a number.
`int( )`	Invalid. Violates Rule 3; a reserved word.
`cosine( )`	Valid. A mnemonic.
`a2b3c4d5( )`	Valid. Not a mnemonic.
`salestax( )`	Valid. A mnemonic.
`amount( )`	Valid. A mnemonic.
`$sine( )`	Invalid. Violates Rule 1; starts with a special character.

3. a.
```
main( )
  {
    input( );    /* input the items purchased */
    salestax( ); /* compute required salestax */
    balance( );  /* determine balance owed */
    calcbill( ); /* determine and output bill */
  }
```

3. b. These functions might be used to determine the billing for an order for goods purchased. The purpose of each function, as indicated by its name, is given in the comment statements (`/* ... */`) for each function call.

Note for Exercises 5 through 9: Many solutions are possible for these exercises. The following are possible answers.

5. Determine the placement of the light fixtures
If you are capable and allowed to do so
Purchase the necessary materials, including the fixtures
and wire the lights in accordance with local laws
Else hire a licensed electrician

7. Determine the courses needed for graduate school
Take the right courses
Maintain an appropriate grade average
Prepare for the graduate record exam (GRE)
Contact engineering graduate schools for admission
interview requirements
Establish contacts for letters of recommendation

9. Select and reserve a camp site
Prepare a list of items to take along
Purchase needed supplies
Reserve a camper at a rental agency (optional)
Arrange for someone to feed plants and watch house
Make arrangements for pet care, if needed
Check and service automobile

Section 1.3

1. a.
```
main( )
  {
    printf("Joe Smith");
    printf("\n99 Somewhere Street");
    printf("\nNonesuch, N.J., 07030");
  }
```

3. a. Six `printf( )` statements would be used.

3. b. One would work, by including newline escape sequences between each two items displayed. Using one line is undesirable since it would make column alignment difficult and the program code hard to debug.

3. c.
```
main( )
{
  printf("DEGREES        RADIANS\n");
  printf("   0           0.0000\n");
  printf("  90           1.5708\n");
  printf(" 180           3.1416\n");
  printf(" 270           4.7124\n");
  printf(" 360           6.2832\n");
}
```

Section 1.4

1 a. Yes.

1 b. It is not in standard form. To make programs more readable and easier to debug, the standard form presented in Section 1.4 of the textbook should be used.

3 a. Two backslashes in a row results in one backslash being displayed.

3 b. `printf("\\ is a backslash.\n");`

Section 2.1

1a. float or double
b. integer
c. float or double
d. integer
e. float or double

3. $1.26e^{2}$ $6.5623e^{2}$ $3.42695e^{3}$ $4.8932e^{3}$ $3.21e^{-1}$ $1.23e^{-2}$ $6.789e^{-3}$

5. Since all of the operands given are integers, the result of each expression is an integer value.

a. 3 + 4 * 6 = 3 + 24 = 27
b. 3 * 4 / 6 + 6 = 12 / 6 + 6 = 2 + 6 = 8
c. 2 * 3 / 12 * 8 / 4 = 6 / 12 * 8 / 4 = 0 * 8 / 4 = 0 / 4 = 0
d. 10 * (1 + 7 * 3) = 10 * (1 + 21) = 10 * 22 = 220
e. 20 − 2 / 6 + 3 = 20 − 0 + 3 = 23
f. 20 − 2 / (6 + 3) = 20 − 2 / 9 = 20 − 0 = 20
g. (20 − 2) / 6 + 3 = 18 / 6 + 3 = 3 + 3 = 6
h. (20 − 2) / (6 + 3) = 18 / 9 = 3
i. 50 % 20 = 10
j. (10 + 3) % 4 = 13 % 4 = 1

7a. 10.0 + 15 / 2 + 4.3 = 10.0 + 7 + 4.3 = 21.3
b. 10.0 + 15.0 / 2 + 4.3 = 10.0 + 7.5 + 4.3 = 21.8
c. 3.0 * 4 / 6 + 6 = 12.0 / 6 + 6 = 2.0 + 6.0 = 8.0
d. 3 * 4.0 / 6 + 6 = 12.0 / 6 + 6 = 2.0 + 6 = 8.0
e. 20.0 - 2 / 6 + 3 = 20.0 − 0 + 3 = 23.0
f. 10 + 17 * 3 + 4 = 10 + 51 + 4 = 65
g. 10 + 17 / 3.0 + 4 = 10 + 5.6666667 + 4 = 19.6666667
h. 3.0 * 4 % 6 + 6 = 12.0 % 6 + 6 = invalid expression
i. 10 + 17 % 3 + 4 = 10 + 2 + 4 = 16

11. `answer1` is the integer 2
`answer2` is the integer 5

19a. 64 * 1024 = 65,536 bytes
b. 128 * 1024 = 131,072 bytes
c. 192 * 1024 = 196,608 bytes
d. 256 * 1024 = 262,144 bytes
e. 64 * 1024 = 65,536 words * 2 bytes/word = 131,072 bytes

f. 64 * 1024 = 65,536 words * 4 bytes/word = 262,144 bytes
g. 360 * 1024 = 368,640 bytes

Section 2.2

1. The following are not valid:

`12345`	does not begin with either a letter or underscore
`while`	reserved word
`$total`	does not begin with either a letter or underscore
`new bal`	cannot contain a space
`9ab6`	does not begin with either a letter or underscore
`sum.of`	contains a special character

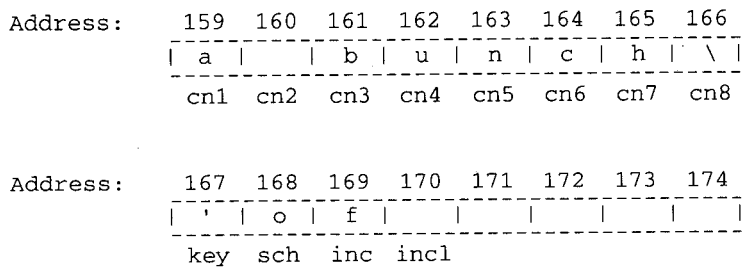
3a. `int count;`
b. `float grade;`
c. `double yield;`
d. `char initial;`

7a.
```
main ( )
  {
    int num1;     /* declare the integer variable num1 */
    int num2;     /* declare the integer variable num2 */
    int total;    /* declare the integer variable total */

    num1 = 25;              /* assign the integer 25 to num1      */
    num2 = 30;              /* assign the integer 30 to num2      */
    total = num1 + num2;    /* assign the sum of num1 and num2 to total */
    printf("The total of %d and %d is %d.\n",num1,num2,total");
    /*      prints:                                    */
    /*      The total of 25 and 30 is 55.  */
  }
```

9.
```
main( )
  {
    int length, width, perim;

    length = 16;
    width = 18;
    perim = length + length + width + width;
    printf("The perimeter is %d.",perim);
  }
```

13. Every variable has a type (e.g. `int`, `float`, etc.) and a value. Additionally, every variable has an address in memory where it is stored.

15a.



Note: The empty locations are usually filled with "garbage" values, meaning their contents are whatever happened to be placed there by the computer or by the previously run program.

Section 2.3

1. `circum = 2.0 * 3.1416 * radius;`

3. `celsius = 5.0 / 9.0 * (fahrenheit - 32.0);`

5. `eltime = totdis / avspeed;`

7. `linexp = linzero * (1.0 + alfpha * (tempfin - tempinit));`

9. `m = x * w * (1 - x) / l;`

11. the average is 26.270000
the average is 682.300000
the average is 1.968000

15.
```
main ( )
{
  int length, width, area;
  area = length * width;
  length = 20;
  width = 15;
  printf("The area is %d", area);
}
```
← this should come after the assignment of values to length and width

The corrected program is:

```
main ( )
{
  int length, width, area;
  length = 20;
  width = 15;
  area = length * width;
  printf("The area is %d", area);
}
```

17. The value of sum is initially set to 311.

```
sum is now 311.
sum is now 311.
sum is now 311.
The final sum is 311.
```

Section 2.4

1 a. `|5|`
b. `|  5|`
c. `|56829|`
d. `| 5.26|`
e. `| 5.27|`
f. `|53.27|`
g. `|534.27|`
h. `|534.00|`

3 a. The comma is within the control string and the statement is not terminated with a semicolon. This statement will generate a compiler error, even if the semicolon is appended to the statement.
b. The statement uses a floating point conversion control sequence with an integer argument. The statement will compile and print an unpredictable result.
c. The statement uses an integer conversion control sequence with a floating point constant. The statement will compile and print an unpredictable result.
d. The statement has no conversion control sequences for the numerical arguments. The statement will compile and print the letters a b c. The constants are ignored.

e. The statement uses a floating point conversion control sequence with an integer argument. The statement will compile and print an unpredictable result.
f. The f conversion character has been omitted from the control string. The statement will compile and print %3.6. The constants have no effect.
g. The formatting string must come before the arguments. The statement will compile and produce no output.

Section 3.1

1 *a.* `sqrt(6.37)`
b. `sqrt(x-y)`
c. `sin(30.0 * 3.1416 / 180.0)`
d. `sin(60.0 * 3.1416 / 180.0)`
e. `fabs(pow(a,2.0) - pow(b,2.0))` or `fabs(a*a + b*b)`
f. `exp(3.0)`

3 *a.*
```
b = sin(x) - cos(x);
```
b.
```
b = pow(sin(x),2.0) - pow(cos(x),2.0);
                or
 b = sin(x) * sin(x) - cos(x) * cos(x);
```
c.
```
area = (c * b * sin(a)) / 2.0;
```
d.
```
c = sqrt(pow(a,2.0) + pow(b,2.0));
                or
  c = sqrt(a*a + b*b));
```
e.
```
p = sqrt(fabs(m-n));
```
f.
```
sum = (a*(pow(r,n) - 1.0)) / (r - 1.0);
```

5.
```
#include <stdio.h>
#include <math.h>
main( )
{
  float dist, x1, y1, x2, y2;

  x1 = 7.0;
  y1 = 12.0;
  x2 = 3.0;
  y2 = 9.0;
  dist = sqrt(pow((x1-x2),2.0) + pow((y1-y2),2.0));
  printf("The distance is %f",dist);
}
```

7.
```
#include <stdio.h>
#include <math.h>
main( )
{
  float height, velocity, theta;

  velocity = 5.0 * 5280.0 / 3600.0 ; /* convert mph to ft/sec */
  theta = 60.0 * 3.1416 / 180.0; /* convert degrees to radians */
  height = 0.5 * pow(velocity,2.0) * pow(sin(theta),2.0) / 32.2 ;
  printf("The maximum height reached is %f feet",height);
}
```

9.
```
#include <stdio.h>
#include <math.h>
main( )
{
  float x, y, r, theta;
```

```
    r = 10.0;
    theta = 30.0 * 3.1416 / 180.0; /* convert to radians */
    x = r * cos(theta);
    y = r * sin(theta);
    printf("The x coordinate is %f\n",x);
    printf("The y coordinate is %f\n",y);
}
```

Section 3.2

1. a. `scanf("%d", &firstnum);`

b. `scanf("%f", &grade);`

c. `scanf("%lf", &secnum);  /* note - the lf is required */`

d. `scanf("%c", &keyval);`

e. `scanf("%d %d %f", &month, &years, &average);`

f. `scanf("%c %d %d %lf %lf",&ch, &num1, &num2, &grade1, &grade2);`

g.
```
scanf("%f %f %f %lf %lf",&interest, &principal, &capital,
                         &price, &yield);
```

h.
```
scanf("%c %c %c %d %d %d",&ch, &letter1, &letter2, &num1,
                          &num2, &num3);
```

i.
```
scanf("%f %f %f %lf %lf %lf",&temp1, &temp2, &temp3,
                             &volts1, &volts2);
```

3. a. Missing & operator in front of num1. The correct form is
```
scanf("%d", &num1);
```

b. Missing & operator in front of `firstnum` and wrong control sequence for price. The correct form is
```
scanf("%d %f %lf", &num1, &firstnum, &price);
```

c. The wrong control sequence for num1 and secnum. The correct form is
```
scanf("%d %f %lf", &num1, &secnum, &price);
```

d. Missing & operators in front of all the variables. The correct form is
```
scanf("%d %d %lf", &num1, &num2, &yield);
```

e. Missing control string entirely. The correct form is
```
scanf("%d %d", &num1, &num2);
```

f. Reversed address and control string. The correct form is
```
scanf("%d", &num1);
```

5 a.
```
#include <stdio.h>
main ( )
{
  float radius, circum;

  printf("Enter the radius of a circle: ");
  scanf("%f", &radius);
  circum = 2.0 * 3.1416 * radius;
  printf("\nThe circumference is %f\n", circum);
}
```

7 a.
```
#include <stdio.h>
main( )
{
  float num1, num2, num3, num4, avg;

  printf("Enter a number: ");
  scanf("%f", &num1);
  printf("\nEnter a second number: ");
```

```
    scanf("%000000f", &num2);
    printf("\nEnter a third number: ");
    scanf("%f", &num3);
    printf("\nEnter a fourth number: ");
    scanf("%f", &num4);
    avg = (num1 + num2 + num3 + num4) / 4.0;
    printf("\nThe average of the four numbers is %f", avg);
  }
```

7 c.
```
#include <stdio.h>
main( )
{
  float number, avg, sum = 0;

  printf("Enter a number: ");
  scanf("%f", &number);
  sum = sum + number;
  printf("\nEnter a second number: ");
  scanf("%f", &number);
  sum = sum + number;
  printf("\Enter a third number: ");
  scanf("%f", &number);
  sum = sum + number;
  printf("\nEnter a fourth number: ");
  scanf("%f", &number);
  sum = sum + number;
  avg = sum / 4.0;
  printf("\nThe average of the four numbers is %f", avg);
}

  avg = (num1 + num2 + num3 + num4) / 4.0;
  printf("\nThe average of the four numbers is %f", avg);
}
```

Section 3.4

1.
```
#include <stdio.h>
#include <math.h>
main( )
{
  float x, y, r, theta;

  r = 10.0;
  theta = 30.0 * 3.1416 / 180.0; /* convert to radians */
  x = r * cos(theta);
  y = r * sin(theta);
  printf("The x coordinate is %f\n",x);
  printf("The y coordinate is %f\n",y);
}
```

3.
```
#include <stdio.h>
#include <math.h>
#define GRAV 32.2
main( )
{
  double time, height;
```

```
    height = 800.0;
    time = sqrt(2.0 * height / GRAV);
    printf("\nIt will take %4.2lf seconds", time);
    printf("\nto fall %7.3lf feet.", height);
}
```

Section 4.1

1. *a.* The relational expression is true. Therefore, its value is 1
   b. The relational expression is true. Therefore, its value is 1
   c. The final relational expression is true. Therefore, its value is 1
   d. The final relational expression is true. Therefore, its value is 1
   e. The final relational expression is true. Therefore, its value is 1
   f. The arithmetic expression has a value of 10
   g. The arithmetic expression has a value of 4
   h. The arithmetic expression has a value of 0
   i. The arithmetic expression has a value of 10

3. *a.* `age == 30`
   b. `temp > 98.6`
   c. `ht < 6.00`
   d. `month == 12`
   e. `let_in == 'm'`
   f. `age == 30 && ht > 6.00`
   g. `day == 15 && month == 1`
   h. `age > 50 || employ >= 5`
   i. `id < 500 && age > 55`
   j. `len > 2.00 && len < 3.00`

Section 4.2

1.a.
```
if (angle == 90)
    printf("The angle is a right angle\n");
else
    printf("The angle is not a right angle\n");
```

b.
```
if (temperature > 100)
    printf("above the boiling point of water\n");
else
    printf("below the boiling point of water\n");
```

c.
```
if (number > 0)
    possum = possum + number;
else
    negsum = negsum + number;
```

d.
```
if (voltage < 0.5)
    flag = 0;
else
    flag = 1;
```

e.
```
if (fabs(volts1 - volts2)< 0.001)
    approx = 0.0;
else
    approx = (volts1 - volts2) / 2.0;
```

f.
```
if (frequency > 60)
    printf("Frequency is too high\n");
```

g.
```
if (fabs(temp1 - temp2) > 2.3)
    error = (temp1 - temp2) * factor;
```

h.
```
if ( x > y && x < 20)
    scanf("%f",&p);    /* this assumes p is a float */
```

i.
```
if (distance > 20 && distance < 35)
   scanf("%f", &time); /* this assumes time is a float */
```

3.
```
#include <stdio.h>
main( )
{
  float voltage;

  printf("Enter a voltage: ");
  scanf("%f",&voltage);
  if (voltage < 12)
    printf("Battery is charging\n");
  else
    printf("Battery is discharging\n");
}
```

5.
```
#include <stdio.h>
main( )
{
  float nyrs;

  printf("Enter the number of years: ");
  scanf("%f", &nyrs);
  if (nyrs < 2)
    printf("The interest rate is 8.5 percent");
  else
    printf("The interest rate is 7 percent");
}
```

7.
```
#include <stdio.h>
main( )
{
  float hours, pay;

  printf("Enter the hours worked: ");
  scanf("%f", &hours);
  if (hours <= 40)
    pay = 8.0 * hours;
  else
    pay = 320.0 + 12.0 * (hours - 40);
  printf("The calculated salary is $%.2f", pay);
}
```

9.
```
#include <stdio.h>
main( )
{
  char status;

  printf("Enter a capital letter: ");
  scanf("%c", &status);
  if (status == 'S')
    printf("The senior engineer's salary is $1000.00\n");
  else
    printf("The junior engineer's salary is $600.00\n");
}
```

11.
```
main ( )
{
  char in_key;
  int position;
```

```
    printf("Enter a lowercase letter: ");
    scanf("%c", &in_key);
    if (in_key >= 'a' && in_key <= 'z')
      printf("The character just entered is a lowercase letter");
    else
      printf("The character just entered is not a lowercase
  letter");
  }
```

Section 4.3

3.
```
#include <stdio.h>
main( )
{
  float angle;

  printf("Enter an angle (in degrees): ");
  scanf("%f", &angle);
  if (angle < 90)
    printf("The angle is acute\n");
  else if (angle > 90)
    printf("The angle is obtuse\n");
  else
    printf("The angle is a right angle\n");
}
```

5.
```
#include <stdio.h>
main ( )
{
  float grade;
  char letter;

  printf("Enter the student's numerical grade: ");
  scanf("%f", &grade);
  if (grade >= 90.0) letter = 'A';
  else if (grade >= 80.0) letter = 'B';
  else if (grade >= 70.0) letter = 'C';
  else if (grade >= 60.0) letter = 'D';
  else letter = 'F';
  printf("\nThe student receives a grade of %c", letter);
}
```

Notice that an `else-if` chain is used. If simple `if` statements were used, a grade entered as 75.5, for example, would be assigned a "C" because it was greater than 60.0. But, the grade would then be reassigned to "D" because it is also greater than 60.0.

7.
```
#include <stdio.h>
main ( )
{
  float fahr,cels,in_temp;
  char letter;

  printf("Enter a temperature followed by");
  printf(" one space and the temperature's type\n");
  printf(" (an f designates a fahrenheit temperature\n");
  printf(" and a c designates a celsius temperature): ");
```

```
    scanf("%f %c", &in_temp, &letter);
    if (letter == 'f' || letter == 'F')
    {
      cels = (5.0/9.0)*(in_temp - 32.0);
      printf("\n%6.2f deg Fahrenheit = %6.2f deg Celsius", in_temp, cels);
    }
    else if (letter == 'c' || letter == 'C')
    {
      fahr = (9.0/5.0)*in_temp + 32.0;
      printf("\n%6.2f deg Celsius = %6.2f deg Fahrenheit", in_temp, fahr);
    }
    else printf("\nThe data entered is invalid.");
  }
```

9. a. This program will run. It will not, however, produce the correct result.

b. and c.

This program evaluates correct incomes for `mon_sales` less than 20000.00 only. If 20000.00 or more were entered, the first `else if` statement would be executed and all others would be ignored. That is, for 20000.00 or more, the income for >= 10000.00 would be calculated and displayed.

Had `if` statements been used in place of the `else if` statements, the program would have worked correctly, but inefficiently (see comments for Exercise 4.b.).

Section 4.4

1.
```
switch (let_grad)
  {
    case 'A':
      printf("The numerical grade is between 90 and 100");
      break;
    case 'B':
      printf("The numerical grade is between 80 and 89.9");
      break;
    case 'C':
      printf("The numerical grade is between 70 and 79.9");
      break;
    case 'D':
      printf("How are you going to explain this one");
      break;
    default:
      printf("Of course I had nothing to do with the grade.");
      printf("\nThe professor was really off the wall.");
  }
```

Section 5.1

1.
```
main ( )
  {
    int count = 2;

    while (count <= 10)
    {
    printf("%d ",count);

    }
  }
```

3 a. Twenty-one items are displayed, which are the integers from1 to 21.
c. Twenty-one items are still displayed, but they would be the integers from 0 to 20 because the `printf( )` call now occurs before the increment.

Section 5.2

3 a.
```
#include <stdio.h>
main ( )
{
  float cels, fahr, incr;
  int num;

  printf("Enter the starting temperature ");
  printf("in degrees Celsius: ");
  scanf("%f", &cels);
  printf("\n\nEnter the number of conversions to be made: ");
  scanf("%d", &num);
  printf("\n\nNow enter the increment between conversions ");
  printf("in degrees Celsius: ");
  scanf("%f", &incr);
  printf("\n\n\nCelsius   Fahrenheit\n");
  printf("----------------------\n");
  while (count <= num)
  {
    fahr = (9.0/5.0)*cels + 32.0;
    printf("%7.2f%15.2f\n", cels, fahr);
    cels = cels + incr;
  }
}
```

7. This program will still calculate the correct values, but the average is now calculated four times. Since it is only the final average that is desired, it is better to calculate the average once, outside of the while loop.

9.
```
#include <stdio.h>
main( )
{
  int count = 1;
  float startmiles, endmiles, gallons, mpg;

  startmiles = 22495;
  while (count <= 8)
  {
    printf("Enter the mileage and gallons: ");
    scanf("%f %f", &endmiles, &gallons);
    mpg = (endmiles - startmiles) / gallons;
    printf("The mpg for this segment of the trip is
%6.2f\n",mpg);
    count++;
    startmiles = endmiles;
  }
}
```

Exercises 5.3

1 a. for (i =1; i <= 20; i++)
b. for (icount = 1; icount <= 20; icount = icount + 2)
c. for (j = 1; j <= 100; j = j + 5)

d. for (icount = 20; icount >= 1; —icount)
e. for (icount = 20; icount >= 1; icount = icount - 2)
f. for (count = 1.0; count <= 16.2; count = count + 0.2)
g. for (xcnt = 20.0; xcnt >= 10.0; xcnt = xcnt − 0.5)

3 a. 10 b. 1024 c. 75 d. −5 e. 40320 f. 0.031250

5.
```
#include <stdio.h>
main( )
{
  int num;

  printf("NUMBER  SQUARE  CUBE\n");
  printf("------  ------  ----\n"0;
  for (num = 0; num <=20; i += 2)
  printf("%3d  %3d  %4d\n", num, num*num, num*num*num);
}
```

7.
```
#include <stdio.h>
main( )
{
  int count;
  float celsius, fahren;

  fahren = 20;
  printf("Fahrenheit  Celsius\n");
  printf("———  ———-\n");
  for (count = 1; count <= 20; ++count)
  {
    celsius = (5.0/9.0) * (fahren - 32);
    printf("  %4.1f    %5.2f\n", fahren, celsius);
    fahren = fahren + 4.0;
  }
}
```

9.
```
#include <stdio.h>
#define COEF 11.7e-6
main( )
{
  int celsius;
  float startlen, starttemp, length;

  startlen = 7365.0;
  starttemp = 0;
  printf("Expansion   Degrees\n");
  printf(" (feet)   (Celsius)\n");
  printf("———  ———-\n");
  for (celsius = 0; celsius <= 40; celsius = celsius + 4)
  {
    length = COEF * startlen * (celsius - starttemp);
    printf(" %6.3f     %2d\n", length, celsius);
  }
}
```

Section 5.4

1.
```
#include <stdio.h>
main( )
{
```

```
    int count;
    float fahren, celsius;

    for(count = 1; count <= 6; ++count)
    {
      printf("\nEnter a fahrenheit temperature: ");
      scanf("%f", &fahren);
      celsius = (5.0/9.0)*(fahren - 32.0);
      printf(" The corresponding celsius temperature is %5.2f\n",celsius);
    }
  }
```

3.
```
#include <stdio.h>
main( )
{
  int conversions, count;
  float gallons, liters;

  printf("Enter the number of conversions to be made: ");
  scanf("%d", &conversions);
  for ( count = 1; count <= conversions; ++count)
  {
    printf("Enter gallon value: ");
    scanf("%f", &gallons);
    liters = 3.785 * gallons;
    printf(" The equivalent liter value is %.3f\n", liters);
  }
}
```

5.
```
#include <stdio.h>
main( )
{
  int i, posnums = 0, negnums = 0;
  float usenum, posavg, negavg;
  float postot = 0, negtot = 0;

  for (i = 1; i <= 5; ++i)
  {
    printf("Enter a number (positive or negative) : ");
    scanf("%f", &usenum);
    if (usenum > 0)
    {
      postot = postot + usenum;
      ++posnums;
    }
    else if (usenum < 0)
    {
      negtot = negtot + usenum;
      ++negnums;
    }
  }
  posavg = postot / posnums;
  negavg = negtot / negnums;
  printf("The positive average is %f\n", posavg);
  printf("The negative average is %f\n", negavg);
}
```

7.
```
#include <stdio.h>
#include <math.h>
main( )
{
  float x, y;

  printf("x value    y value\n");
  printf("-----    -------\n");
  for ( x = 5.0; x < 10.0; x = x + 0.2)
  {
    y = 3.0 * pow(x,5.0) - 2.0 * pow(x,3.0) + x;
    printf("%6.2f    %10.2f\n", x, y);
  }
}
```

Section 5.5

1.
```
main( )
{
  int i, j;
  float total, avg, data;

  for (i = 1; i <= 4; ++i)
  {
    printf("Enter 6 results for experiment #%d: ",i);
    for (j = 1, total = 0.0; j <= 6; ++j)
    {
      scanf("%f", &data);
      total += data;
    }
    avg = total/6;
    printf("  The average for experiment #%d is %.2f\n\n", i, avg);
  }
}
```

Note: When entering data for each experiment, the six individual results may be entered on one line with a space between each entry, on six individual lines, or any combination of these.

2.
```
main( )
{
  int bowler, game;
  float score, plyr_tot, plyr_avg, team_tot, team_avg;

  for(bowler = 1, team_tot = 0; bowler <= 5; ++bowler)
  {
    for(game = 1, plyr_tot = 0; game < 3; ++game)
    {
      printf("Enter the score for bowler %d game %d: ",bowler,game);
      scanf("%f",&score);
      plyr_tot = plyr_tot + score;
    }
    team_tot = team_tot + plyr_tot;
    plyr_avg = plyr_tot/3.0;
```

```
      printf("  The average for bowler %d is %5.2f\n",bowler,plyr_avg);
    }
    team_avg = team_tot/15.0;
    printf("The average for the whole team is %5.2f",team_avg);
  }
```

Section 5.6

3 a.
```
main( )
  {
    int num, digit;

    printf("Enter an integer: ");
    scanf("%d", &num);
    printf("\nThe number reversed is: ");
    do
    {
      digit = num % 10;
      num /= 10;
      printf("%d", digit);
    } while (num > 0);
  }
```

Section 6.1

1 a. `float voltages[10];`
b. `float temps[50];`
c. `char code[30];`
d. `int years[100];`
e. `float velocity[32];`
f. `float distance[1000];`
g. `int code[6];`

3 a. `scanf("%d %d %d",&grades[0], &grades[2], &grades[6]);`
b. `scanf("%f %f %f",&volts[0], &volts[2], &volts[6]);`
c. `scanf("%f %f %f",&amps[0], &amps[2], &amps[6]);`
d. `scanf("%d %d %d",&dist[0], &dist[2], &dist[6]);`
e. `scanf("%f %f %f",&velocity[0], &velocity[2], &velocity[6]);`
f. `scanf("%f %f %f",&time[0], &time[2], &time[6]);`

5 a. `a[1] a[2] a[3] a[4] a[5]`
b. `a[1] a[3] a[5]`
c. `b[3] b[4] b[5] b[6] b[7] b[8] b[9] b[10]`
d. `b[3] b[6] b[9] b[12]`
e. `c[2] c[4] c[6] c[8] c[10]`

7.
```
#include <stdio.h>
main( )
{
  int temp[8], sum, i;
  float average;

  sum = 0;  /* initialize here or in the declaration */
  for( i = 0; i < 7; ++i)
  {
    printf("Enter a value for element number %d: ", i);
```

```
    scanf("%d", &temp[i]);
    sum = sum + temp[i];
  }
  printf("\nThe values stored in the array are:\n");
  for (i = 0; i < 7; ++i)
    printf(" %d", temp[i]);
  average = sum / 8.0;
  printf("\nThe average of these values is %f\n", average);
}
```

Section 6.2

1 a. `int grades[10] = {89, 75, 82, 93, 78, 95, 81, 88, 77, 82};`
b. `double amount[5] = {10.62, 13.98, 18.45, 12.68, 14.76};`
c. `double rates[100] = {6.29, 6.95, 7.25, 7.35, 7.40, 7.42};`
d. `float temp[64] = {78.2, 69.6, 68.5, 83.9, 55.4, 67.0, 49.8, 58.3, 62.5, 71.6};`
e. `char code[15] = {'f', 'j', 'm', 'q', 't', 'w', 'z'};`

3.
```
#include <stdio.h>
main( )
{
  float slopes[9] = {17.24, 25.63, 5.94,
                     33.92, 3.71, 32.84,
                     35.93, 18.24, 6.92};
  int i;
  float max = 0.0, min = 999.9;

  for(i = 0; i < 8; ++i)
  {
    if (slopes[i] <min) min = slopes[i];
    if (slopes[i] > max) max = slopes[i];
  }
  printf("\nThe minimum array value is %5.2f",min);
  printf("\nThe maximum array value is %5.2f",max);
}
```

5.
```
char goodstr1[13] = {'G', 'o', 'o', 'd', ' ',
                     'M', 'o', 'r', 'n', 'i', 'n', 'g'};

char goodstr1[] = {'G', 'o', 'o', 'd', ' ',
                   'M', 'o', 'r', 'n','i', 'n', 'g'};

char goodstr1[] = "Good Morning";
```

Note: This last declaration creates an array having one more character than the first two. The extra character is the null character.

Section 6.3

1 a. `int array[6][10];`
b. `int codes[2][5];`
c. `char keys[7][12];`
d. `char letter[15][7];`
e. `double vals[10][25];`
f. `double test[16][8];`

3.
```
#include <stdio.h>
main( )
{
```

```
    int i, j, total = 0;
    int val[3][4] = {8,16,9,52,3,15,27,6,14,25,2,10};

    for (i = 0; i < 3; ++i)
      for (j = 0; j < 4; ++j)
        total = total + val[i][j];
    printf("The total of the values is %d\n", total);
  }
```

5.
```
#include <stdio.h>
main( )
{
  int i, j, total = 0;
  int max = -999;
  int val[4][5] = {16, 22, 99, 4, 18,
                  -258, 4, 101, 5, 98,
                   105, 6, 15, 2, 45,
                    33, 88, 72, 16, 3};

  for (i = 0; i < 3; ++i)
    for (j = 0; j < 4; ++j)
      if (val[i][j] > max) max = val[i][j];
  printf("The maximum array value is %d\n", max);
}
```

Section 7.1

1 ***a.*** `factorial( )` expects to receive one integer value and returns an integer value.
b. `price( )` expects to receive one integer and two double precision values, in that order, and returns a double.
c. An `int` and two double precision values, in that order, must be passed to `yield( )`. A double precision value is returned.
d. A character and two floating point values, in that order must be passed to `interest( )`. A character value is returned.
e. Two floating point values, in that order, must be passed to `total( )`. The function returns a floating point value.
f. Two integers, two characters, and two floating point values, in that order, must be passed to `roi( )`. The function returns a floating point value.
g. Two integers and two character values, in that order, are expected by `get_val( )`. No value is returned by the function.

3 ***a.***
```
int factorial(int);
```
b.
```
double price(int, double, double);
```
c.
```
double yield(int, double, double);
```
d.
```
char interest(char, float, float);
```
e.
```
float total(float, float);
```
f.
```
float roi(int, int, char, char, float, float);
```
g.
```
void get_val(int, int, char);
```

5 ***a.*** The `find_abs( )` function is included within the larger program written for Exercise 5b.

5 ***b.***
```
#include <stdio.h>
main( )
{
  double dnum, absval;
  double find_abs(double); /* function prototype */
```

```
  printf("\nEnter a number: ");
  scanf("%lf", &dnum); /* the %lf is required for doubles */
  absval = find_abs(dnum);
  printf("The absolute value of %lf is %lf",dnum, absval);
}

double find_abs(double num)
{
  double aval;

  if (num < 0) aval = -num;
    else aval = num;
  return(aval);
}
```

7 ***a.*** The `sqr_it( )` function is included within the larger program written for Exercise 7b.

b.

```
#include <stdio.h>
main( )
{
  double first, sqrfirst;
  double sqr_it(double);

  printf("Please enter a number: ");
  scanf("%lf",&first); /* %lf must be used in scanf for a double */
  sqrfirst = sqr_it(first);
  printf("The square of %f is %f\n", first, sqrfirst);
}

double sqr_it(double num)
{
  return(num * num);
}
```

9 ***a.*** The function for producing the required table is included within the larger program written for Exercise 9b.

b.

```
#include <stdio.h>
main( )
{
  void table(void);
  table( );   /* call the table( ) function */
}

void table(void)
{
  int num;

  printf("NUMBER  SQUARE  CUBE\n");
  printf("------  ------  ----\n");

  for (num = 1; num < 10; ++num)
    printf("%3d     %3d     %4d\n", num, num*num, num*num*num);
}
```

Section 7.2

1. `double sort_arr(double in_array[500])`

or

`double sort_arr(double in_array[])`

3. `float prime(float rates[256])` or `float prime(float rates[])`

5.
```
#include <stdio.h>
main( )
{
  float rates[9] = {6.5, 7.2, 7.5, 8.3, 8.6,
                    9.4, 9.6, 9.8, 10.0};
  void show(float rates[]); /* function prototype */

  show(rates);
}

void show(float rates[])
{
  int i;

  printf("The elements stored in the array are:");
  for(i = 0; i < 8; ++i)
    printf("\n %4.1f", rates[i]);
}
```

7.
```
#include <stdio.h>
main( )
{
  double price[10] = {10.62, 14.89, 13.21, 16.55, 18.62,
                      9.47, 6.58, 18.32, 12.15, 3.98};
  double quantity[10] = {4.0, 8.5, 6.0, 7.35, 9.0,
  double amount[10];    15.3, 3.0, 5.4, 2.9, 4.8};
  int i;
  void extend(double [], double [], double []); /* prototype */

  extend(price, quantity, amount); /* call the function */
  printf("The elements in the amount array are:");
  for(i = 0; i < 9; ++i)
    printf("\n %7.31f", amount[i]);
}

void extend(double prc[], double qnty[], double amt[])
{
  int i;

  for(i = 0; i < 9; ++i)
    amt[i] = prc[i] * qnty[i];
}
```

Section 7.4

1. a.

Variable name	Data type	Scope
price	integer	global to main( ), roi( ), and step( )
years	long integer	global to main( ), roi( ), and step( )
yield	double-precision	global to main( ), roi( ), and step( )
bondtype	integer	ocal to main( ) only
interest	double-precision	local to main( ) only
coupon	double-precision	local to main( ) only
count	integer	local to roi( ) only
eff_int	double-precision	local to roi( ) only
numofyrs	nteger	local to step( ) only
fracpart	float	local to step( ) only

Note: Although arguments of each function assume a value which is dependent on the calling function, these arguments can change values within their respective functions. This makes them behave as is they were local variables within the called function.

Section 7.5

1. a. Local variables may be automatic, static, or register. It is important to realize, however, that not all variables declared inside of functions are necessarily local. An example of this is is an external variable.

b. Global variables may be static or external.

3. The first function declares `yrs` to be a static variable and assigns a value of one to it only once, when the function is compiled. Thereafter, each time the function is called the value in `yrs` is increased by two. The second function also declares `yrs` to be static, but assigns it the value one every time it is called, and the value of `yrs` after the function is finished will always be three. By resetting the value of `yrs` to one each time it is called, the second function defeats the purpose of declaring the variable to be static.

5. The scope of a variable means where in the program the variable is recognized and can be used within an expression. If, for example, the variable years is declared inside a function, it is local and its scope is inside that function only. If the variable is declared outside of any function, it is global and its scope is anywhere below the declaration but within that file, unless another file of the same program declares that same variable to be external.

Section 8.1

1 a. On an IBM or IBM-compatible personal computer (PC, XT, or AT), a file name may have up to eight characters, and optionally a decimal point followed by three more characters. If a string is used to hold the file name, an extra character should be provided for the NULL, for a total of 13 characters. All UNIX systems permit file names of 14 characters.

3.
```
out_file = fopen("math.dat", "w");
book = fopen("book.dat", "w");
resfile = fopen("resist.dat", "w");
exfile = fopen("exper2.dat", "w");
pfile = fopen("prices.cat", "w");
ratefile = fopen("rates.mem","w");
```

5 a. The data may be entered in a variety of ways. One possibility is to enter the data, line by line, and write each line to a file as was done in Exercise 1a. A second method is to

use a text editor to write the data to a file. A third possibility is to enter the data as individual items for each line, assemble the items into a complete string, formatted as desired, and then write the string out. A fourth possibility is to enter the data as individual items and write the file as individual items. The program below uses the last approach.

```
#include <stdio.h>
main( )
{
  float data[] = {16.25, 18.96, 22.34, 18.94, 17.42, 22.63};
  FILE *outfile;
  int i;

  outfile = fopen("result.dat", "w");
  for ( i = 0; i < 5; ++i)
    fprintf(outfile,"%f\n",data[i]);
  fclose(outfile);
}
```

b. The approach taken in version 1 of this program is to input data from the file as individual data items using a `for` loop. Version 2 uses the same approach using a `while` loop.

Version 1:

```
#include <stdio.h>
main( )
{
    float val, sum, average;
    int count;
    FILE *infile;

    infile = fopen("result.dat", "r");
    for(count = 0; count < 5; ++count)
    {
      fscanf(infile, "%f", &val);  /* read one value */
      printf("The value just read is %f\n", val);
      sum = sum + val;   /* add the value to the sum */
    }
    average = sum / 6.0;
    printf("The sum of the values read is %f\n", sum);
    printf("The average of the values read is %f\n", average);
    fclose(infile);
  }
```

Version 2:

```
#include <stdio.h>
main( )
{
    float val, sum, average;
    int count;
    FILE *infile;

    infile = fopen("result.dat", "r");
    count = 0;
    sum = 0;
    while (fscanf (infile, "%f", &val) != EOF)
    {
      printf("The value just read is %f\n", val);
```

```
        count = count + 1;  /* add one to the count */
        sum = sum + val;    /* add the value to the sum */
    }
    average = sum / count;
    printf("The sum of the values read is %f\n", sum);
    printf("The average of the values read is %f\n", average);
    fclose(infile);
}
```

7 *a.* The data may be entered in a variety of ways. One possibility is to enter the data, line by line, and write each line to a file as was done in Exercise 1a. A second method is to use a text editor to write the data to a file. A third possibility is to enter the data as individual items for each line, assemble the items into a complete string, formatted as desired, and then writeout the string. A fourth possibility is to enter the data as individual items and write the file as individual items. The program below uses the last approach construction of an in-memory string.

```
#include <stdio.h>
main( )
{
    int carnum[5] = { 54, 62, 71, 85, 97 };
    int miles[5] = {250, 525, 123, 1322, 235 };
    int gallons[5] = { 19, 38, 6, 86, 14 };
    int i;
    FILE *outfile;

    outfile = fopen("cars", "w");
    for(i = 0; i < 4
    ; ++i)
    fprintf(outfile, "%d %d %d\n", carnum[i], miles[i],
gallons[i]);
    fclose(outfile);
}
```

b. The approach taken in version 1 of this program is to input data from the file as individual data items using a `for` loop. Version 2 uses the same approach using a `while` loop.

Version 1:

```
#include <stdio.h>
main( )
{
    int carnum, miles, gallons;
    float totmiles = 0.0, totgals = 0.0, mpg, avgmpg;
    int count;
    FILE *infile;

    infile = fopen("cars", "r");
    for(count = 0; count < 4; ++count)
    {
      fscanf(infile, "%d %d %d", &carnum, &miles, &gallons);
      mpg = miles / gallons;
      printf("%2d %4d %2d %5.2f\n", carnum, miles, gallons, mpg);
      totmiles = totmiles + miles;
      totgals = totgals + gallons;
    }
    avgmpg = totmiles / totgals;
    printf("The total miles driven was %f\n", totmiles);
```

```
    printf("The total gallons used was %f\n", totgals);
    printf("The average miles per gallon is %f\n", avgmpg);
    fclose(infile);
  }
```

Version 2:

```
#include <stdio.h>
main( )
{
    int carnum, miles, gallons;
    float totmiles = 0.0, totgals = 0.0, mpg, avgmpg;
    FILE *infile;

    infile = fopen("cars", "r");
    while(fscanf(infile,"%d %d %d",&carnum,&miles,&gallons)!=EOF)
    {
      mpg = miles / gallons;
      printf("%2d %4d %2d %5.2f\n", carnum, miles, gallons, mpg);
      totmiles = totmiles + miles;
      totgals = totgals + gallons;
    }
    avgmpg = totmiles / totgals;
    printf("The total miles driven was %f\n", totmiles);
    printf("The total gallons used was %f\n", totgals);
    printf("The average miles per gallon is %f\n", avgmpg);
    fclose(infile);
  }
```

Section 8.2

1. The `fseek( )` function call moves the character pointer to the last character in the file, which is the EOF character at offset position 12. The `ftell( )` function reports the offset of the character currently pointed to. This is the EOF character. Thus, a 12 is returned by `ftell( )`.

5.

```
#include <stdio.h>
main( )
{
  FILE *fopen( ), *in;
  char fname[13];

  printf("\nEnter a file name: ");
  scanf("%s", fname);
  in = fopen(fname, "r");
  fseek(in,0L,2); /* move to the end of the file */
  printf("There are %ld characters in the file.\n",ftell(in));
  fclose(in);
}
```

Section 8.4

1 a.
```
in_file = fopen("test.dat","rt");
```
b.
```
descrip = fopen("descri","wt");
```
c.
```
out_file = fopen("names", "at");
```
d.
```
disktype = fopen("types","rb");
```
e.
```
idcodes = fopen("codes","wb");
```
f.
```
balances = fopen("balance.dat","ab");
```

3.
```
#include <stdio.h>
main( )
{
  int i;
  float number, total, average;
  float nums[4] = {92.65, 88.72, 77.46, 82.93};
  FILE *test;

  test = fopen("numdata","wt");
  for (i = 0; i < 4; ++i) /* write 4 lines to the file */
  fprintf(test,"%f\n", nums[i]);
  printf("The file has been written.\n");
  fclose(test);
  test = fopen("numdata", "rt"); /* open for reading */
  total = 0.0; /* this can be moved to the declaration */
  for (i = 0; i < 4; ++i) /* now read the file */
  {
    fscanf(test,"%f", &number);
    total = total + number;
  }
  average = total / 4.0;
  printf("The total of the numbers in the file is %.2f\n", total);
  printf("The average of theses numbers is %.2f\n", average);
  fclose(test);
}
```

5.
```
#include <stdio.h>
main( )
{
  int i, j;
  float x1, y1, x2, y2;
  float slope, midxpt, midypt;
  float nums[3][4] = { 6.3, 8.2, 18.25, 24.32,
                       4.0, 4.0, 10.0, -5.0,
                      -2.0, 5.0, 4.0, 5.0 };
  FILE *coord;

  coord = fopen("points","wt");
  for (i = 0; i < 3; ++i)
  {
    for (j = 0; j < 4; ++j)
      fprintf(coord,"%f ", nums[i][j]);
    printf("\n");   /* start a new line in the file */
  }
  printf("The file has been written.\n");
  fclose(coord);
  coord = fopen("points", "rt"); /* open for reading */
  while(fscanf(coord,"%f %f %f %f",&x1,&y1,&x2,&y2)!=EOF)
  {
    slope = (y2 - y1) / (x2 - x1);
    midxpt = (x1 + x2) / 2.0;
    midypt = (y1 + y2) / 2.0;
    printf("Slope %d = %.3f\n", i, slope);
    printf("Midpoint coordinates %d = (%.3f,%.3f)\n",i,midxpt,midypt);
  }
  fclose(coord);
}
```

Section 9.1

1. The solution is $x = 3, y = -1$

3. The modified program is:

```
#define EPSILON 0.001
#include <stdio.h>
#include <math.h>
main( )
{
  float det2(float, float, float, float); /* prototype */
  float a1, b1, k1, a2, b2, k2, x, y, valdet;

  printf("Enter a1, b1, k1, a2, b2, k2: ");
  scanf("%f %f %f %f %f %f", &a1,&b1,&k1,&a2,&b2,&k2);
  valdet = det2(a1,b1,a2,b2);
  if ( fabs(valdet) >= EPSILON)
  {
    x = det2(k1,b1,k2,b2) / valdet;
    y = det2(a1,k1,a2,k2) / valdet;
    printf("\nThe solution is:\n");
    printf("x = %f\n", x);
    printf("y = %f\n", y);
  }
  else
    printf("\nA unique solution does not exist.\n");
}
float det2(float a, float b, float c, float d)
/* function to evaluate the 2 x 2 determinant  | a b | */
/*                                             | c d | */
{
  float valdet2;
  valdet2 = a * d - b * c;
  return(valdet2);
}
```

5 a.

```
#include <stdio.h>
main( )
{
  float det2(float, float, float, float); /* prototype */
  float a1, b1, k1, a2, b2, k2, valdet;
  float vg, vb, rg, rb, rl, i1, i2;

  printf("Enter vb in volts: ");
  scanf("%f", &vb);
  printf("Enter rg, rb, and rl in ohms: ");
  scanf("%f %f %f", &rg, &rb, &rl);
  a1 = rg + rb;
  b1 = -rb;
  a2 = -rb;
  b2 = rb + rl;
  valdet = det2(a1,b1,a2,b2); /* do this once, outside of loop */
  printf(" VG      I1       I2\n");
  printf("(volts)   (amps)    (amps)\n");
  printf("-------   ------    ------\n");
  for( vg = 12.5; vg < 15.0; vg = vg + 0.1)
  {
     k1 = vg -vb;
```

```
        k2 = vb;
        i1 = det2(k1,b1,k2,b2) / valdet;
        i2 = det2(a1,k1,a2,k2) / valdet;
        printf(" %4.1f    %5.2f    %5.2f\n", vg, i1, i2);
    }
}
float det2(float a, float b, float c, float d)
/* function to evaluate the 2 x 2 determinant | a b | */
/*                                            | c d | */
{
  float valdet2;
  valdet2 = a * d - b * c;
  return(valdet2);
}
```

7 a. The optimum straight line fit is y = 0.25 + 0.4x

9. The optimum parabolic fit is

$$y = 4.25 - 1.75\,x + 0.25\,x^2$$

Section 9.2

The results of some of the root finding exercises are dependent on the roundoff errors generated. Therefore different results may occur when the programs are run on your own computer.

1 a. In Program 9-4 replace the assignment statement for y by

```
y = x ** 4 + 4 * x ** 3 - 7 * x ** 2 - 22 * x + 24
```

The roots are at x = −4, x = −3, x = 1, and x = 2 .

b. In Program 9-5 replace the assignment statement for y as done in Exercise 1a. The four roots are displayed.

3. In Program 9-7 replace the return expression in the `f( )` function by

```
pow(x,4.0) + 4 * pow(x,3.0) - 7 * pow(x,2.0) - 22 * x + 24
```

For a step size of 1 and 0.5 no roots are displayed, since the roots are integers and either x_1 or x_2 fall very close to the roots. Roundoff errors may result in the values of y_1 and y_2 having the same sign, in which case the root is not displayed. All four roots are displayed for the step sizes of 0.499, 0.1, and 0.01. A step size of 0.499 will ensure that all values of x_1 and x_2 do not fall very close to the roots. For smaller step sizes accumulated roundoff error makes it unlikely that values of x_1 and x_2 are very close to the roots.

5. In Program 9-7 replace the return expression in the `f()` function by

```
cos(x)
```

The six roots

x = 1.57, 4.71, 7.85, 11.00, 14.14, and 17.28

are displayed for each of the four step sizes.

7. In Program 9-8 replace the return expression in the `f( )` function by

```
cos(x)
```

The six roots are located by selecting the left and right bounds of x shown below. The number of iterations required is also shown.

Left and right bounds of x	Root	Number of iterations
0 to 3	1.571	12
3 to 6	4.712	18
6 to 9	7.854	13
9 to 12	10.995	12
12 to 16	14.137	12
16 to 20	17.279	13

9. In Program 9-9 replace the return expression in the `f( )` function by

```
cos(x)
```

When the same bounds of x used in Exercise 7 are entered, the following results are obtained.

Left and right bounds of x	Root	Number of iterations
0 to 3	1.571	3
3 to 6	4.712	3
6 to 9	7.853	3
9 to 12	10.996	4
12 to 16	14.137	3
16 to 20	20.420	5

The first five roots are located much more rapidly than for the bisection method. However, the last run located a root outside the interval `(at x = 13 * pi / 2)`. If the last run is repeated, with 16 and 18 entered as the left and right bounds of x, the root at `x = 17.279` is then located after 3 iterations.

Section 9.3

1 a. In Program 9-10 replace the return expression in the `f( )` function by

```
pow(x,3.0) + 2 * pow(x,2.0) + 3 * x + 1
```

The results of the four runs are shown below.

N	AREA
2	2.0625
10	3.1225
50	3.3569
1000	3.4136

b. In Program 9-11 replace the return expression in the `f( )` function as done in Exercise 1a. The results of the four runs are shown below.

N	AREA
2	3.3438
10	3.4138
50	3.4165
1000	3.4166

3. The modified program is:

```
#include <stdio.h>
#include <math.h>
```

```
main( )
{
  int i, n;
  float x, a, b, width, area, error;
  float f(float);  /* function prototype */

  printf("Enter the left and right bounds of x (a b): ");
  scanf("%f %f", &a, &b);
  printf(" N  Approximate Area  Percent Error\n");
  printf("--- ----------------  -------------\n");
  for ( n = 1; n < 10; ++n)
  {
    width = (b - a) / n;
    x = a;
    area = 0;
    for ( i = 1; i < n; ++i)
    {
      area = area + width * f(x + width/2.0);
      x = x + width;
      }
      error = (area - 3.14159) / area * 100;
      printf("%2d    %f     %f\n", n, area, error);
    }
  }
  /* the function is evaluated below */
  float f(float x)
  {
    return( sqrt(4.0 - pow(x,2.0)));
  }
```

The following table is displayed when the program is run.

```
Enter the left and right bounds of x (a b): 0 2
 N     Approximate Area    Percent Error
--     ----------------    -------------
 1         3.464102           9.310107
 2         3.259367           3.613507
 3         3.206413           2.021654
 4         3.183929           1.329786
 5         3.171988           0.958327
 6         3.164767           0.732338
 7         3.160012           0.582973
 8         3.156687           0.478248
 9         3.154254           0.401488
10         3.152411           0.343261
```

5. Use Program 9-11 to calculate the area under the three required curves. The results of the three runs are shown below.

N	x_c	y_c
5	0.8194	0.2545
10	0.8299	0.2718
50	0.8332	0.2775

The value of x_c converges more rapidly than does the value of y_c. This results from the fact that the calculation of x_c requires approximating the area under a fifth order polynomial, while the calculation of y_c involves an eighth order polynomial. The higher order polynomial presents a steeper curve which results in a poorer approximation of area by the mod-

ified rectangular method. Nevertheless the error in y_c is only about 2 % for N = 10 and less than 0.1 % for N = 50.

Section 10.1

1. `&average` means "the address of the variable named average."

3a.
```
#include <stdio.h>
main( )
{
  char key, choice;
  int num, count;
  long date;
  float yield;
  double price;

  printf("The address of the variable key is %p\n",&key);
  printf("The address of the variable choice is %p\n",&choice);
  printf("The address of the variable num is %p\n",&num);
  printf("The address of the variable count is %p\n",&count);
  printf("The address of the variable date is %p\n",&date);
  printf("The address of the variable yield is %p\n",&yield);
  printf("The address of the variable price is %p\n",&price);
}
```

5a. `*x_addr`
b. `*y_addr`
c. `*pt_yld`
d. `*pt_miles`
e. `*mptr`
f. `*pdate`
g. `*dist_ptr`
h. `*tab_pt`
i. `*hours_pt`

7a. Each of these variables are pointers. This means that addresses will be stored in each of these variables.

7b. They are not very descriptive names and do not give an indication that they are pointers.

9. All pointer variable declarations must have an asterisk. Therefore, c, e, g, and i are pointer declarations.

11.
```
Variable: pt_num
Address:  500
          ---------------
          |     8096    |
          ---------------
Variable: z_addr
Address: 8024
          ---------------
          |    20492    |
          ---------------
Variable: pt_day
Address: 14862
          ---------------
          |    20492    |
          ---------------
```

```
Variable: amt_addr
Address: 564
          ---------------
          |    16256    |
          ---------------
Variable: num_addr
Address: 10132
          ---------------
          |    18938    |
          ---------------
Variable: pt_yr
Address: 15010
          ---------------
          |      694    |
          ---------------
```

```
Variable: years
Address: 694
         ---------------
         |    1987     |
         ---------------

Variable: amt
Address: 16256
         ---------------
         |     154     |
         ---------------

Variable: balz
Address: 20492
         ---------------
         |      25     |
         ---------------

Variable: m
Address: 8096
         ---------------
         |             |
         ---------------

Variable: firstnum
Address: 18938
         ---------------
         |     154     |
         ---------------

Variable: k
Address: 24608
         ---------------
         |     154     |
         ---------------
```

Section 10.2

1. a. double *price;
b. int *minutes;
c. char *key;
d. double *yield;

3.
```
#include <stdio.h>
main ( )
{
  int firstnum, secnum, max;
  void find_max (int, int, int *);   /* function prototype  */
  printf("Enter a number: ");
  scanf("%d", &firstnum);
  printf("\nGreat! Please enter a second number: ");
  scanf("%d", &secnum);
  find_max(firstnum, secnum, &max);
  printf("\n\nThe maximum of the two numbers is %d.", max);
}

void find_max(int x, int y, int *max_addr)
{
  if (x >= y)
     *max_addr = x;
  else
     *max_addr = y;
  return;
}
```

Notice that `find_max` is declared in the function `main( )` as void because we're not returning any value; rather there is a "return by reference." In other words, no value is being passed back to `main( )`; instead a memory location's content known to both `main( )` and `find_max( )` is being altered.

5.
```
#include <stdio.h>
main ( )
{
  int hours, mins, sec, totsecs;
  void secs (int, int, int, int *);  /* prototype */
  printf("Enter the time as hh mm ss: ");
  scanf("%d %d %d", &hours, &mins, &sec);
  secs(hours, mins, sec, &totsecs);
  printf("\nThe total seconds is %d", totsecs);
}
```

```
void secs(int hr, int min, int sec, int *total)
{
  *total = hr * 3600 + min * 60 + sec;
  return;
}
```

7.
```
#include <stdio.h>
main( )
{
  int mon, day, yr;
  long int days;
  void datefind (long, int *, int *, int *);
  printf("Enter the days: ");
  scanf("%ld", &days);
  datefind(days, &yr, &mon, &day);   /* to an integer */
  printf("\nThe date is %2d/%2d/%4d\n", mon, day, yr);
}

void datefind(long days, int *yr, int *mon, int *day)
{
  *yr = (int) (days/360);
  *mon = (int) (days - *yr * 360)/30;
  *day = days - (*yr * 360) - (*mon * 30);
  return;
}
```

9. In `main( )` the variables `min` and `hour` refer to integer quantities, while in `time( )` the variables `min` and `hours` are pointers to integers. Thus, there are four distinct variables in all, two of which are known in `main( )` and two of which are known in `time( )`. The computer (actually, the compiler) keeps track of each variable with no confusion. The effect on a programmer, however, may be quite different.

When in `main( )` the programmer must remember to use the names `min` and `hour` as integer variables. When in `time( )` the programmer must "switch" viewpoints and use the same names as pointer variables. Debugging such a program can be quite frustrating because the same names are used in two different contexts. It is, therefore, more advisable to avoid this type of situation by adopting different names for pointers than those used for other variables. A useful "trick" is to either prepend each pointer name with a `pt_` notation or append each pointer name with `_addr`.

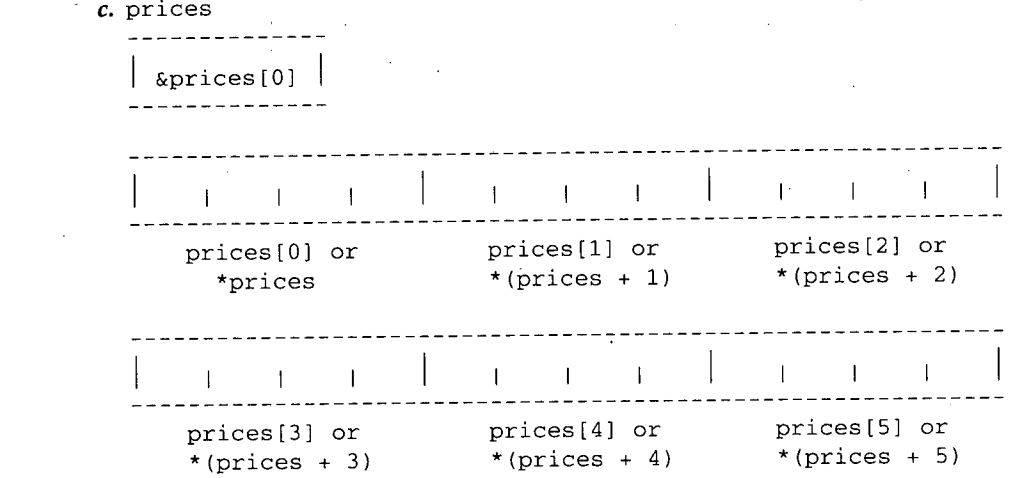
Section 10.3

1. a. `*(prices + 5)`
b. `*(grades + 2)`
c. `*(yield +10)`
d. `*(dist + 9)`
e. `*mile`
f. `*(temp + 20)`
g. `*(celsius + 16)`
h. `*(num + 50)`
i. `*(time + 12)`

3a. The declaration `double prices [5];` causes storage space for five double precision numbers, creates a pointer constant named prices, and equates the pointer constant to the address of the first element `(&prices[0])`.

b. Each element in prices contains four bytes and there are five elements for a total of 20 bytes.

c.



d. The byte offset for this element, from the beginning of the array, is 3 * 4 = 12 bytes.

5.
```
#include <stdio.h>
main( )
{
  static float rates[] = {12.9, 18.6, 11.4, 13.7, 9.5, 15.2, 17.6};
  int i;

  printf("The elements of the array are:\n");
  for(i = 0; i < 6; ++i)
    printf("\n%5.2f", *(rates + i)); /* The variable pointed to by */
}                                    /* rates offset by i          */
```

Section 10.4

1.
```
#include <stdio.h>
main( )
{
  int nums[5] = {16, 54, 7, 43, -5};
  int total, *n_pt;

  n_pt = &nums[0];  /* n_pt = nums; is equivalent */
  printf("The elements in the array are: ");
  for( ; n_pt < (nums + 4); ++n_pt)
    printf("%d ", *n_pt);
}
```

3. a.
```
#include <stdio.h>
main( )
{
  static char strng[] = "Hooray for all of us";
  char *mess_pt;

  mess_pt = &strng[0];  /* mess_pt = strng; is equivalent */
  printf("The elements in the array are: ");
  for( ; *mess_pt != '\0'; ++mess_pt)
    printf("%c", *mess_pt);
}
```

b.
```
#include <stdio.h>
main( )
{ static char strng[] = "Hooray for all of us";
  char *mess_pt;

  mess_pt = &strng[0];  /* mess_pt = strng; is equivalent */
  printf("The elements in the array are: ");
  while (*mess_pt != '\0') /* search for the null character */
    printf("%c", *mess_pt++);
}
```

Section 10.5

1.
```
int sort_arr(double in_array[500])
int sort_arr(double in_array[])
int sort_arr(double *in_array)
```

3.
```
int prime(float rates[256])
int prime(float rates[])
int prime(float *rates)
```

5. The problem to this method of finding the maximum value lies in the line

if(max < *vals++) max = *vals;

This statement compares the correct value to `max`, but then increments the address in the pointer before any assignment is made. Thus, the element assigned to max by the expression `max = *vals` is one element beyond the element pointed to within the parentheses.

9. a. The following output is obtained:

```
33
16
99
34
```

This is why:

```
*(*val) = *(val[0]) = val[0][0] = 33;
*(*val + 1) = *(val[1]) = val[1][0] = 16;
*(*(val + 1) + 2) = *(*(val[1]) + 2) = *(val[1][2]) = 99;
*(*val) + 1 = *(val[0]) + 1 = val[0][0] + 1 = 33 + 1 = 34.
```

In other words, for any two dimensional `array, arr[x][y]`, what we really have are two levels of pointers. What's meant by `*(arr + x)` is that there are x number of pointers, each successively pointing to `arr[1][0], arr[2][0], arr[3][0], ..., arr[x][0]`. So an expression such as `*(*(arr + x) + y)` translates to `arr[x][y]`.

Section 10.6

1. If the filename is an internal pointer name, the definition of `p_file( )` is:

```
p_file(FILE *fname)
```

If the file name is an external file name, which can be stored in an array of characters (see Exercise 3), the definition of `p_file` is:

```
p_file(char [])
```

3. The `fcheck( )` function is included below with a driver function used to test it,

```
#include <stdio.h>
main( )  /* driver function to test fcheck( ) */
{
  int fcheck( );
  char name[13];

  printf("Enter a file name: ");
  scanf("%s", name);
  if(fcheck(name) == 1)
    printf("The file exists and can be opened.");
  else
    printf("The file cannot be opened - check that it exists.");
}

int fcheck(char *fname)
{

  if(fopen(fname, "r") == 0)
    return(0);
  else
    return(1);
}
```

Note: *The* **`fcheck( )`** *function performs essentially the same check as* **`fopen( )`**, *except* **`fopen( )`** *returns a* **`non_zero`** *pointer instead of an integer value.*

Section 11.1

1b.

```
#include <stdio.h>
main( )
{
  char line[81];
  void vowels(char []); /* function prototype */

  printf("Enter a string.\n");
  gets(line);
  vowels(line);
}
void vowels(char strng[])
{
  int i = 0, v = 0;  /* Array element number = i; vowel counter = v */
  char c;
  while((c = strng[i++]) != '\0')
  switch(c)
  {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
    putchar(c);
    ++v;
  }          /* continued on next page */
```

```
    putchar('\n');
    printf("There were %d vowels.", v);
    return;
}
```

3a. The function is included in the program written for Exercise 3b.

3b.
```
#include <stdio.h>
main( )
{
  char strng[81];
  void count_str(char []);  /* function prototype */

  printf("Enter a line of text\n");
  gets(strng);
  count_str(strng);
}

void count_str(char message[])
{
  int i;

  for(i = 0; message[i] != '\0'; ++i);  /* The semicolon at the end */
                                        /* of this statement is the */
                                        /* null statement           */
  printf("\nThe number of total characters, including blanks,");
  printf("\n the line just entered is %d.\n", i);
  return;
}
```

7.
```
#include <stdio.h>
main( )
{
  char word[81];
  void del_char(char [], int, int); /* function prototype */

  printf("Enter a string\n");
  gets(word);
  printf("\n%s\n",word);
  del_char(word, 13, 5); /* string, how many to delete, starting position */
  puts(word);            /* display the edited string */
  }

  void del_char(char strng[], int x, int pos)
  {
    int i, j;
    i = pos-1;  /* first element to be deleted (actually, overwritten) */
    j = i + x;  /* first element beyond delete range */
    while (strng[j] != '\0')
      strng[i++] = strng[j++]; /* copy over an element */
    strng[i] = '\0';  /* close off the edited string */
    return;
  }
```

This program assumes the number of characters to be deleted actually exists. Otherwise the `while` loop would not terminate (unless it just happened to encounter another null character somewhere in memory beyond the original string).

9a. The `to_upper( )` function is included in the program written for Exercise 9.c.

9c.
```
#include <stdio.h>
main( )
{
  char strng[81];
  int i = 0;
  char to_upper(char); /* function prototype */

  printf("Enter a line of text\n");
  gets(strng);
  while (strng[i] != '\0') /* get the character */
  {
    strng[i] = to_upper(strng[i]); /* send it to the function */
    ++i;                          /* move to next character */
  }
  printf("The string, with all lower case letters converted is:\n");
  puts(strng);
}

char to_upper(char ch)
{

    if ( ch >= 'a' && ch < 'z')  /* test it */
      return(ch - 'a' + 'A');   /* change it, if necessary */
    else
      return(ch);
}
```

11.
```
#include <stdio.h>
main( )
{
  char strng[81];
  int i = 0, count = 1;

  printf("Enter a line of text\n");
  gets(strng);
  if(strng[i] == ' ' || strng[i] == '\0')
    --count;
  while(strng[i] != '\0')
  {
    if(strng[i] == ' ' && (strng[i + 1] != ' ' && strng[i + 1] != '\0'))
      ++count;   /* encountered a new word */
    ++i;       /* move to the next character */
  }
  printf("\nThe number of words in the line just entered is %d", count);
}
```

The program increases the word count whenever a transition from a blank to a non-blank character occurs. Thus, even if words are separated by more than one space the word count will be incremented correctly. Initially the program assumes the text starts with a word (count = 1). If the first character is either a blank or an end-of-string `Null`, this assumption is incorrect and the count is decremented to zero.

Section 11.2

1a.
```
*text = 'n'
*(text + 3) = ' '
*(text + 10) = ' '
```

b.
```
*text = 'r'
*(text + 3) = ' '
*(text + 10) = 'h'
```

c.
```
*text = 'H'
*(text + 3) = 'k'
*(text + 10) = 'o'
```

d.
```
*text = 'T'
*(text + 3) = 'p'
*(text + 10) = 'd'
```

3.
```
#include <stdio.h>
main( )
{
  char line[81];
  void vowels(char *);

  printf("Enter a string.\n");
  gets(line);
  vowels(line);
}

void vowels(char *strng) /* strng is treated as a pointer variable */
{
  int v = 0;  /* v = vowel counter */
  char c;

  while((c = *strng++) != '\0') /* an address is incremented */
  switch(c)
  {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
      putchar(c);
      ++v;
}
  putchar('\n');
  printf("There were %d vowels.", v);
  return;
}
```

5.
```
#include <stdio.h>
main( )
{
  char strng[81];
  void count_str(char *);  /* function prototype */

  printf("Enter a line of text\n");
  gets(strng);
  count_str(strng);
}

void count_str(char *message) /* message as a pointer variable */
{
  int count;

  for(count = 0; *message++ != '\0'; ++count) ; /* The semicolon at the */
                                                /* end of this statement*/
                                                /* is the null statement*/
  printf("\nThe number of total characters, including blanks,");
```

```
    printf("\n in the line just entered is %d.\n", count);
    return;
  }
```

7.
```
#include <stdio.h>
main( )
{
  char forward[81], rever[81];
  void reverse(char *, char *); /* two pointer arguments */

  printf("Enter a line of text:\n");
  gets(forward);
  reverse(forward,rever);
  printf("\n\nThe text: %s \n",forward);
  printf("spelled backwards is: %s \n",rever);
}

void reverse(char *forw, char *rev) /* pointers to characters */
{
  int i = 0;

  while(*(forw + i) != '\0')     /* count the elements */
    ++i;                        /* in the string   */
  for(-i; i >= 0; -i)
    *rev++ = *(forw + i);
  *rev = '\0';                  /* close off reverse string */
  return;
}
```

9. The function is included within a complete program.

```
#include <stdio.h>
main( )
{
  char ch, line[81];
  void append_c(char *, char); /* pass a pointer and a character */

  printf("Enter a line of text: ");
  gets(line);
  printf("Enter a single character: ");
  ch = getchar();
  append_c(line,ch);
  printf("The new line of text with the appended last character is:\n");
  puts(line);
}

void append_c(char *strng, char c)
{

    while(*strng++ != '\0')  /* this advances the pointer */
        ;                    /* one character beyond '\0 '*/
    --strng;              /* point to the '\0') */
    *strng++ = c;          /* replace it with the new char */
    *strng = '\0';         /* close the new string   */
}
```

13.
```
trimrear(char *strng)
  {

    while(*strng != '\0') ++strng;  /* move to end of string  */
     --strng;                       /* move to char before '\0' */
    while(*strng == ' ') --strng;   /* skip over blank characters */
    *(++strng) = '\0';              /* close off string */
    return;
  }
```

Section 11.3

1.
```
char *text = "Hooray!";
char test[] = {'H','o','o','r','a','y','!','\0'};
```

3. `message` is a pointer constant. Therefore, the statement `++message`, which attempts to alter its address, is invalid. A correct statement is

```
putchar(*(message + i);
```

Here the address in `message` is unaltered and the character pointed to is the character offset i bytes from the address corresponding to `message`.

Section 11.4

1 ***a.***
```
!four score and ten!  /* field width specifier is ignored */
```
b.
```
!              Home!!
```
c.
```
!Home!              !
```
d.
```
!Ho                 !
```
e.
```
!                 Ho!
```

3.
```
#include <stdio.h>
main( )
{
  char strn[30];
  float num1, num2, num3;
  void separate(char *, float *, float *, float *); /* prototype */

  printf("Enter three numbers on the same line,");
  printf("\n separating the numbers with one or more spaces: ");
  gets(strn);   /* read the numbers in as a string */
  separate(strn, &num1, &num2, &num3);
  printf("The three numbers are %f %f %f",num1, num2, num3);
}

void separate(char *st_addr, float *n1_addr, float *n2_addr, float *n3_addr)
{
  sscanf(st_addr,"%f %f %f",n1_addr, n2_addr, n3_addr);
  return;
}
```

Functions like `separate( )` are useful when reading data from a file. Rather than read individual items sequentially, a complete line of the file is read in as a string and then dissembled internally within the program. This isolates any line that does not have the required number and types of data items.

```
5. #include <stdio.h>
   main( )
   {
     char strng1[80], strng2[100];
     int num1, num2;
     void combine(char *, char *, int, int); /*function prototype */

     printf("Enter a string: ");
     gets(strng1);
     printf("Enter an integer number: ");
     scanf("%d",&num1);
     printf("Enter a second integer number: ");
     scanf("%d",&num2);
     combine(strng1, strng2, num1, num2);
     printf("A string containing all inputs is:\n");
     puts(strng2);
   }

   void combine(char *source, char *dest, int n1, int n2)
   {
     sprintf(dest,"%s %d %d",source, n1, n2); /* write the string */
     return;
   }
```

Functions like `combine( )` are useful in assembling separate data items into a single line for output to a file. The file will then contain identically formatted lines, each line containing the same number and types of data items. Additionally, the file will be in ASCII, which can easily be read by any word processing program, for easy inspection external to the program that created it.

Section 12.1

1. a.
```
struct s_temp
  {
    int id_num;
    int credits;
    float avg;
  };
```

b.
```
struct s_temp
  {
    char name[40];
    int month;
    int day;
    int year;
    int credits;
    float avg;
  };
```

c.
```
struct s_temp
  {
    char name[40];
    char street[80];
    char city[40];
    char state[2];
    char zip[5];   /* int zip will not store a leading 0 */
  };
```

d.
```
struct s_temp
{
  char name[40];
  float price;
  char date[8];    /* Assumes a date in the form xx/xx/xx */
};
```

e.
```
struct s_temp
{
  int part_no;
  char desc[100];
  int quant;
  int reorder;
};
```

3 *a.*
```
#include <stdio.h>
main( )
{
  struct
  {
    int month;
    int day;
    int year;
  } date;    /* define a structure variable named date */


  printf("Enter the current month: ");
  scanf("%d", &date.month);
  printf("Enter the current day: ");
  scanf("%d", &date.day);
  printf("Enter the current year: ");
  scanf("%d", &date.year);
  printf("\n\nThe date entered is %d/%d/%d.",
          date.month, date.day, date.year);
}
```

b.
```
#include <stdio.h>
main( )
{
  struct clock
  {
  int hours;
  int minutes;
  int seconds;
  } time;       /* define a structure variable named time */

  printf("Enter the current hour: ");
  scanf("%d", &time.hours);
  printf("Enter the current minute: ");
  scanf("%d", &time.minutes);
  printf("Enter the current second: ");
  scanf("%d", &time.seconds);
  printf("\n\nThe time entered is %02d:%02d:%02d", time.hours,
          time.minutes, time.seconds);
}
```

Note the use of the conversion sequence %02d. The 0 forces the field of 2 to be filled with leading zeros.

```
5. #include <stdio.h>
   main( )
   {
     struct
     {
       int hours;
       int minutes;
     } time;

     printf("Enter the current hour: ");
     scanf("%d", &time.hours);
     printf("Enter the current minute: ");
     scanf("%d", &time.minutes);
     if(time.minutes != 59)
       time.minutes += 1;
     else
     {
       time.minutes = 0;
       if(time.hours != 12)
         time.hours += 1;
       else
         time.hours = 1;
     }
     printf("\nThe time in one minute will be %02d:%02d",
             time.hours, time.minutes);
   }
```

Note the use of the conversion sequence %02d. The 0 forces the field of 2 to be filled with leading zeros.

Section 12.2

```
1. a. struct s_temp
      {
        int id_num;
        int credits;
        float avg;
      };

      main( )
      {
        struct s_temp student[100];
   b. struct s_temp
      {
        char name[40];
        int month;
        int day;
        int year;
        int credits;
        float avg;
      };

      main( )
      {
        struct s_temp student[100];
```

c.
```
struct s_temp
  {
    char name[40];
    char street[80];
    char city[40];
    char state[2];
    int zip;       /* or char zip[5]; */
  };

  main()
  {
    struct s_temp address[100];
```

d.
```
struct s_temp
  {
    char name[40];
    float price;
    char date[8];    /* Assumes a date in the form xx/xx/xx */
  };
  main( )
  {
    struct s_temp stock[100];
```

e.
```
struct s_temp
  {
    int part_no;
    char desc[100];
    int quant;
    int reorder;
  };
  main( )
  {
    struct s_temp inven[100];
```

3.
```
#include <stdio.h>
struct mon_days
{
 char name[10];
 int days;
};

main( )
{
   struct mon_days convert[12] =
                      { "January", 31,"February", 28,
                        "March", 31, "April", 30,
                        "May", 31, "June", 30,
                        "July", 31, "August", 31,
                        "September", 30, "October", 31,
                        "November", 30, "December", 31
                      };
  int i;

  printf("\nEnter the number of a month: ");
  scanf("%d", &i);
  printf("\n%s has %d days.", convert[i-1].name, convert[i-1].days);
}
```

Section 12.3

1.
```
#include <stdio.h>
struct date
{
  int month;
  int day;
  int year;
};

main( )
{
  struct date present;
  long num;
  long days(struct date); /* argument is a date structure */

  printf("Enter the month: ");
  scanf("%d", &present.month);
  printf("Enter the day: ");
  scanf("%d",&present.day);
  printf("Enter the year: ");
  scanf("%d", &present.year);
  num = days(present);
  printf("\nThe number of days since the turn of the century is %ld", num);
}

long days(struct date temp)
{
  return(temp.day + 30*(temp.month - 1) + 360*temp.year);
}
```

***Note:** The pointer version of the function* **`long days( )`** *is written for Exercise 3.*

3.
```
#include <stdio.h>
struct date
{
  int month;
  int day;
  int year;
};

main( )
{
  struct date present;
  long num;
  long days(struct date *); /* argument is a pointer to a date structure */

  printf("Enter the month: ");
  scanf("%d", &present.month);
  printf("Enter the day: ");
  scanf("%d", &present.day);
  printf("Enter the year: ");
  scanf("%d", &present.year);
  num = days(&present);
  printf("\n\nThe number of days since the turn of the century is %ld", num);
}
```

```
long days(struct date *temp)
{
  return(temp->day + 30*(temp->month - 1) + 360*temp->year);
}
```

5.
```
#include <stdio.h>
static struct date
{
  int month;
  int day;
  int year;
};

main( )
{
  struct date present;
  long num;
  long days(struct date);

  printf("Enter the date as mm/dd/yy: ");
  scanf("%d/%d/%d", &present.month, &present.day, &present.year);
  num = days(present);
  printf("\nThe number of days since the turn of the century is %ld", num);
}

long days(struct date temp)
{
  long act_days;
  int daycount[12] = { 0, 31, 59, 90, 120, 151,
                       180, 211, 241, 271, 302, 333};
  act_days = temp.day + daycount[temp.month-1] + 364*temp.year;
  return(act_days);
}
```

Section 12.4

1.
```
#include <stdio.h>
struct tele_typ
{
  char name[30];
  char phone_no[15];
  struct tele_typ *nextaddr;
};

main( )
{
  struct tele_typ t1 = {"Acme, Sam", "(201) 898-2392"};
  struct tele_typ t2 = {"Dolan, Edith", "(213) 682-3104"};
  struct tele_typ t3 = {"Lanfrank, John", "(415) 718-4518"};
  struct tele_typ *first;
  char strng[30];
  void search(struct tele_typ *, char *); /* function prototype */

  first = &t1;
  t1.nextaddr = &t2;
  t2.nextaddr = &t3;
```

```
    t3.nextaddr = NULL;
    printf("Enter a name: ");
    gets(strng);
    search(first, strng);
  }

  void search(struct tele_typ *contents, char *strng)
  {

    printf("\n%s",strng);
    while(contents != NULL)
    {
      if(strcmp(contents->name,strng) == 0)
      {
        printf("\nFound. The number is %s.", contents->phone_no);
        return;
      }
      else
      {
        contents = contents->nextaddr;
      }
    }
    printf("\nThe name is not in the current phone directory.");
  }
```

3. To delete the second record, the pointer in the first record must be changed to point to the third record.

5 a.

```
struct phone_bk
    {
      char name[30];
      char phone_no[15];
      struct phone_bk *previous;
      struct phone_bk *next;
    };
```

Section 12.5

1. The `check( )` function is included below in a complete program used to verify that `check( )` works correctly.

```
#include <stdio.h>
struct tel_typ
{
  char name[25];
  char phone_no[15];
  struct tel_typ *nextaddr;
};

main( )
{
  int i;
  struct tel_typ *list, *current;
  char *malloc( );
  void display(struct tel_typ *);
```

```
  void populate(struct tel_typ *);
  void check(struct tel_typ *);

  list = (struct tel_typ *) malloc(sizeof(struct tel_typ));
  check(list);
  current = list;
  for(i = 0; i < 2; ++i)
  {
    populate(current);
    current->nextaddr = (struct tel_typ *) malloc(sizeof(struct tel_typ));
    check(current->nextaddr);
    current = current->nextaddr;
  }
  populate(current);
  current->nextaddr = NULL;
  printf("\nThe list consists of the following records:\n");
  display(list);
}

void check(struct tel_typ *addr)
{

  if(addr == NULL)
  {
    printf("No available memory remains. Program terminating");
    exit(0); /* Function to stop program and return to operating system */
  }
  else
    return;
}       /* continued on next page */

void populate(struct tel_typ *record)
{
  printf("\nEnter a name: ");
  gets(record->name);
  printf("Enter the phone number: ");
  gets(record->phone_no);
  return;
}

void display(struct tel_typ *contents)
{
  while(contents != NULL)
  {
    printf("\n%-30s %-20s", contents->name, contents->phone_no);
    contents = contents->nextaddr;
  }
  return;
}
```

3. The `insert( )` function is included below in a complete program used to verify that `insert( )` works correctly. As written, the function will insert a structure after the structure whose address is passed to it. Since the address of the first structure is passed to it, the new structure is inserted between the first and second structures.

Note that if the populate function call is removed from the insert function then `insert( )` becomes a general insertion program that simply creates a structure and

correctly adjusts the address members of each structure. Also notice the notation used in `insert( )`. The expression

```
addr->nextaddr->nextaddr
```

is equivalent to

```
(addr->nextaddr)->nextaddr
```

This notation was not used in `main( )` because the pointer variable current is first used to store the address in `list->nextaddr` using the statement:

```
current = list->nextaddr;
```

The statement:

```
current->nextaddr = NULL;
```

in `main( )` however, could have been written as:

```
list->nextaddr->nextaddr = NULL;
```

An interesting exercise is to rewrite `main( )` such that the pointer variable named current is removed entirely from the function.

```
#include <stdio.h>
struct tel_typ
{
  char name[25];
  char phone_no[15];
  struct tel_typ *nextaddr;
};
/* the following are global function prototypes - each function */
/* expects a pointer to a structure of type tel_typ */
void display(struct tel_typ *);
void populate(struct tel_typ *);
void insert(struct tel_typ *);

main( )
{
  int i;
  struct tel_typ *list, *current;
  char *malloc();

  list = (struct tel_typ *) malloc(sizeof(struct tel_typ));
  populate(list); /* populate the first structure */
  list->nextaddr = (struct tel_typ *) malloc(sizeof(struct tel_typ));
  current = list->nextaddr;
  populate(current); /* populate the second structure */
  current->nextaddr = NULL;
  printf("\nThe list initially consists of the following records:");
  display(list);
  insert(list);  /* insert between first and second structures */
  printf("\nThe new list now consists of the following records:");
  display(list);
}
```

```
void insert(struct tel_typ *addr)
{
  struct tel_typ *temp;
  char *malloc( );

  temp = addr->nextaddr;  /* save pointer to next structure */
  /* now change the address to point to the inserted structure */
  addr->nextaddr = (struct tel_typ *) malloc(sizeof(struct tel_typ));
  populate(addr->nextaddr); /* populate the new structure */
  /* set the address member of the new structure to the saved addr */
  addr->nextaddr->nextaddr = temp;
  return;
}

void populate(struct tel_typ *record)
{
  printf("\nEnter a name: ");
  gets(record->name);
  printf("Enter the phone number: ");
  gets(record->phone_no);
  return;
}

void display(struct tel_typ *contents)
{
  while(contents != NULL)
  {
    printf("\n%-30s %-20s", contents->name, contents->phone_no);
    contents = contents->nextaddr;
  }
  return;
}
```

5. The `modify( )` function is included below in a complete program used to verify that `modify( )` works correctly. The driver function creates a single structure, populates it, and then calls `modify( )`. `modify( )` itself calls the function `repop( )`. An interesting extension is to write `repop( )` such that an ENTER key response retains the original structure member value.

```
#include <stdio.h>
struct tel_typ
{
  char name[25];
  char phone_no[15];
  struct tel_typ *nextaddr;
};
/* the following are global function prototypes - each function */
/* expects a pointer to a structure of type tel_typ */
void modify(struct tel_typ *);
void populate(struct tel_typ *);
void repop(struct tel_typ *);
void display(struct tel_typ *);

main( )
{
  int i;
  struct tel_typ *list;
  char *malloc( );
```

```
  list = (struct tel_typ *) malloc(sizeof(struct tel_typ));
  populate(list); /* populate the first structure */
  list->nextaddr = NULL;
  modify(list);  /* modify the structure members */
}

void populate(struct tel_typ *record)
{
  printf("\nEnter a name: ");
  gets(record->name);
  printf("Enter the phone number: ");
  gets(record->phone_no);
  return;
}

void modify(struct tel_typ *addr)
{

  printf("\nThe current structure members are:");
  display(addr);
  repop(addr);
  printf("\nThe structure members are now:");
  display(addr);
  return;
}

void repop(struct tel_typ *record)
{
  printf("\n\nEnter a new name: ");
  gets(record->name);
  printf("Enter a new phone number: ");
  gets(record->phone_no);
  return;
}

void display(struct tel_typ *contents)
{
  while(contents != NULL)
  {
    printf("\n%-30s %-20s", contents->name, contents->phone_no);
    contents = contents->nextaddr;
  }
  return;
}
```

Section 12.6

1. `printf( )` function calls, with the correct control sequences are contained within the following program.

```
#include <stdio.h>
union
{
  float rate;
  double taxes;
  int num;
```

```
} flag;
main( )
{
  flag.rate = 22.5;
  printf("\nThe rate is %f",flag.rate);
  flag.taxes = 44.7;
  printf("\ntaxes are %f",flag.taxes);
  flag.num = 6;
  printf("\nnum is %d",flag.num);
}
```

5. Since a value has not been assigned to `alt.btype`, the display produced is unpredictable (the code for a 'y' resides in the storage locations overlapped by the variables `alt.ch` and `alt.btype`). Thus, either a garbage value will be displayed or the program could even crash.

Chapter 13

1a.	*1b.*	*1c.*
11001010	11001010	11001010
&10100101	|10100101	^10100101
10000000	11101111	01101111

3a. 0157 = 001 101 111; 001 101 111 < 1 = 011 011 110 = 0336
b. 0701 = 111 000 001; 111 000 001 < 2 = 100 000 100 = 0404
c. 0873 = undefined, there is no octal digit higher than 7.
d. 087 = undefined.

5a. The binary number `00001100`, which equals octal 014, is the required mask pattern.
b. Two zeros could be placed in the third and fourth positions to reproduce the flag bits in these positions. However, the inclusive `OR` operation cannot set the remaining bits to zero.
c. The binary number `00001100`, which equals octal 014, is the required mask pattern.

7. Any character ASCII binary code that has `bit six` equal to zero will be unaffected by the conversion Program 12-2. These include all the characters listed in Appendix F having either a one-digit hexadecimal code or a two-digit hexadecimal code beginning in either a 1, 4, or 5. Conversely, any character whose ASCII binary code has bit six equal to a one will be affected. This includes all the characters listed in Appendix F having a two-digit hexadecimal code beginning in either 2, 3, 6, or 7.

9.
```
#include <stdio.h>
main( )
{
  char word[81];   /* enough storage for a complete line */
  void lower(char *); /* function prototype */

  printf("Enter a string of both upper and lowercase letters:\n");
  gets(word);
  printf("\nThe string of letters just entered is:\n");
  puts(word);
  lower(word);
  printf("\nThis string, in lowercase letters is:\n");
  puts(word);
}
```

```
void lower(char *word)
{
  while (*word != '\0')
    if (*word >= 'A' && *word < 'Z')
      *word++ |= 0X20;
    else
      ++word;
}
```

11.
```
#include <stdio.h>
main( )
{
  char message[81];   /* enough storage for a complete line */
  FILE *outfile;
  void encrypt(char *); /* function prototype */

  printf("Enter a sentence:\n");
  gets(message);
  printf("\nThe sentence just entered is:\n");
  puts(message);
  encrypt(message);
  outfile = fopen("coded.dat","w");
  fputs(message,outfile);
  printf("\nThe encrypted message has been written to coded.dat\n");
  fclose(outfile);
}

void encrypt(char *message)
{
  while (*message != '\0')
    *message++ ^= 52;
}
```

13.
```
#include <stdio.h>
main( )
{
  char ch;
  void showbits(char); /* function prototype */

  printf("\nType in a character: ");
  ch = getchar( );
  printf("\nThe character just read is: %c", ch);
  printf("\n\nThe binary code of this character is: ");
  showbits(ch);
}

void showbits(char ch)
{
  int i, mask = 0X80;

  for (i = 1; i < 8; ++i)
  {
    /* print a 1 if the next bit is 1, else print a 0 */
    if( (mask & ch) > 0 )
      printf("%d ",1);
```

```
        else
          printf("%d ", 0);
        mask >>= 1; /* shift the mask to inspect the next bit */
      }
    }
```

Exercises 14.1

1 a. The value of the expression is 6.
b. The value of the expression is 6.
c. The value of the expression is 13.
d. The value of the expression is 3.
e. The value of the expression is 0.
f. The value of the expression is 0.
g. The value of the expression `a + b > 20` is a + 0 = 2.
h. The value of the expression `num = a + b > 20` is 2.
i. The value of the expression `a || b` is `2 || 3`, which is `1(True)`.
j. The value of the expression `num = a || b` is the same as the value assigned to num, which is 1.

3.
```
c = amount*rate;
b = c;
a = b;
```

5 a.
```
flag = (a >= b) ? 1 : 0
```

b.
```
flag = (a == b || c == d) ? 1 : 0
```

Exercises 14.3

1 a.
```
#define NEGATE (x) -(x)
```

1 b.
```
#include <stdio.h>
#define NEGATE(x) -(x)
main( )
{
  float num;

  printf("Enter a number: ");
  scanf("%f",.&num);
  printf("\nThe negative of %f is %f.", num, NEGATE(num));
}
```

3 a.
```
#define PI 3.1416
#define CIRCUM(x) 2.0 * PI * (x) b.
```

3 b.
```
#include <stdio.h>
#define PI 3.1416
#define CIRCUM(x) 2.0 * PI * (x)
main( )
{
  float radius;

  printf("Enter a radius: ");
  scanf("%f", &radius);
  printf("\nThe circumference is %f.", CIRCUM(radius));
}
```

5 a.
```
#define MAX(x,y) ((x) >= (y)) ? (x) : (y)
```

5 b.
```
include <stdio.h>
#define MAX(x,y) ((x) >= (y)) ? (x) : (y)
main( )
{
  float num1, num2;

  printf("Enter two numbers, separated by at least a space: ");
  scanf("%f %f", &num1, &num2);
  printf("\nThe largest number entered is %f", MAX(num1,num2));
}
```

Exercises 14.4

1 a. The following program opens the file, reads each character, and displays it.

```
#include <stdio.h>
main(int argc, char *argv[])
{
  FILE *in_file;
  char cc;

  in_file = fopen(argv[1],"r");
  while( (cc = getc(in_file)) != EOF)
    putchar(cc);
  fclose(in_file);
}
```

Note: Instead of reading each character until the end-of-file is reached, the `fgets( )` function could have been used to read in a line at a time. Since `fgets( )` returns a `NULL` when it encounters the end-of-file sentinel, the appropriate statement would be

```
while (fgets(line, 81, in_file) != NULL);
```

If this statement is used, line would have to be declared as

```
char line[81];
```

To output a line at a time either the `puts( )` or `fputs( )` functions can be used. The `puts( )` function adds its own newline escape sequence at the end of each line; the `fputs( )` function does not.

b. The program will open and display the contents of any file. The file can, therefore, be either a data or a program file.

Index

A

abs() function, 99
accumulating, 58–59
accuracy, numerical, 149
acid rain, 128–30
actual arguments, 281
addition, 38
addition operator (+), 38, 39
addresses
  arrays as, 439–45
  of byte, 28
  decrementing, 435–36
  incrementing, 435–36
  in linked lists, 501–03
  passing, 137, 421–26, 439
  in pointer constants, 451
  in pointers, 434–38
  storing, 94–95, 413
  use of subscript to obtain, 428–33
  using, 94, 414, 439
  of variables, 43–44, 92, 411, 429
address operator (&), 94, 412
algorithms, 4–6
  coding, 6
  definition of, 4
  determining, 75–76
  selection of, 77–80
  allowable error, 373
American Standard Code for Information Interchange. *See* ASCII
AND bit operator (&) 517–520
AND logical operator (&&), 146
a.out, 543, 547
Applications
  acid rain, 128–30
  averase and RMS valves, 402–05
  battery charger, 360–64
  coin toss simulation, 300–02
  curve plotting, 259–63
  data scaling, 263–66
  data validation, 181–83
  dynamic storage allocation, 505–11
  linked lists, 498–505
  master file update, 343–45
  numerical integration, 388–402
  pendulum clocks, 83–85
  pollen counts, 340–43
  quadratic equations, 177–80
  root finding, 369–87
  sine approximation, 130–34
  telephone switching networks, 85–87
argc, 545–46
argument(s), 16, 36, 278, 281
  actual, 281
  command line, 543–47
  formal, 278
  of the function, 16
argument declarations, 277–79
argv[], 544–46
arithmetic, with pointers, 434–38
arithmetic operators, 35–36, 532–33
  associativity, 39
  conversion rules for, 102–03
  integer division, 38
  negation, 38
  precedence, 38–39, 103
  simple, 35
  summary of operators, 38
arithmetic right shift, 528
array(s)
  as addresses, 439–45
  component of, 242
  element of, 242
  global, 250–51, 254
  index value, 243–44, 247
  larger-dimensional, 257–58
  local, 250, 254
  one-dimensional, 241–48
  passing, 291–93
  pointer, 471–74
  single-dimensional, 241–48
  static, 251
  of structures, 487–89
  two-dimensional, 253–56, 442–43
array initialization, 250–51, 489
array names, 241
  as pointers, 428–33
array values, input and output of, 245–48
ASCII (American Standard Code for Information Interchange), 33
  character codes, 33, 355, 519
  escape sequence codes, 42
assignment operators, 54–55, 57–58, 62, 238
assignment statements, 44, 53–55
  accumulating statements, 58–60
  counting statements, 60–62
associativity, 38–39, 148
asterisks, for comments, 22

auto storage class, 310–11
auto variables, 310
average value, 402–05

B

backslashes (/), 19,34
 in macros, 542
bar symbol (1), 71
BASIC (Beginners All-purpose Symbolic Instruction Code), 4
battery charger, 360–64
BCPL language C, 4
bell control codes, 355–56
binary files, 348, 350
binary operators, 517
binary search, 377
binary subtraction, 35
bisection method, 377
bit, 26
bit operators, 517–29
 AND operator, 396–401, 517–20
 complement operator (NOT), 524
 different-size data items, 525
 exclusive OR operator, 522–24
 inclusive OR operator, 520–22
 shift operators, 526–28
B language, 4
body of function, 14, 278
Borland's Turbo C compiler, 559–62
bounds check, 245, 450
braces, 14
 in compound statements, 152
 in function, 14, 24
 in main() function, 14
 in two-dimensional arrays, 254
break statement, 212
bubble sort, 269, 272
buffer, 119
buffered input, scanf() with, 119–21
bug, 77, 193
bytes, 26

C

call by reference, 421
call by value, 421
called function, 276
calling functions, 276
case, 172
case sensitive, 12
cast, 357, 102–03, 507
cast operator, 103
C compiler(s), 6
 Borland's Turbo C, 559–62
 Microsoft Quick C, 563–66
char, 60, 97
character
 nonprintable, 355
 null, 252, 332, 454, 544
 printable, 354
character-by-character input, 458–59
character codes, 28, 33
character constant, 33
character type, 33
character variables, 47
C language, 4
clear control code, 355–56
closing a file, 328
COBOL (COmmon Business Oriented Language), 3
coding, 6, 138
 algorithm, 4–6
coin toss simulation, 300–02
command line arguments, 543–47
command mode, 556
commas
 errors in using, 239
 in for statement, 220
 in arguments, 36
 in two-dimensional arrays, 254
comments, 20–23
complied language, 8
compiler, 8
compile-time errors, 189
complement operator (~), 524–25
component of array, 242
compound statements, 152–53
coumputer program, 2, 6
 coding algorithm to, 6
 modular, 10–14
 translation of, 6
conditional expressions, 533–34
conditional operator (?:), 534
conditions, 145. *See also* relational expressions
constants
 named, 122–25
 symbolic, 123
continue statement, 212
control codes, 354–56
CONTROL keys, 120
conversion control sequence, 36, 47, 93
 %c, 37
 %d, 36–37, 67–68
 %f, 37, 67–68
 %1f, 47

%s, 456, 470, 475, 547
%u, 947 fn
control string, 36, 107–08
counting statement, 60
curve plotting, 259–63
C preprocessor, 124
compiling and linking, 556–58
Borland's Turbo C, 559–62
Microsoft Quick C, 563–66
editing, 554–56
C statement, 55

D

data encryption, 523–24
data file, 324
closing, 328
declaring, 324–25
opening, 325–26
reading and writing, 327–34
data scaling, 263–66
data type, 36, 31, 103
character, 33
double precision, 31
enumerated, 535–37
escape sequences, 34
exponential notation, 32
floating point, 31
integers, 31
unions, 512–13
user-specified, 535–39
void, 279
data validation, 181–83
debugging, 77, 186, 193
decimal numbers, 71, 355
declaration of functions, 280
declarations of pointers, 96, 415
declaration statement, 45–48
as definition statement, 49–51
extern in, 314–16
within main () function, 45
placement of, 45
decrement operator (--), 60–61, 137
default, 172
#define statement, 122–24, 210
defining macros, 539–42
definition statement,
desk checking, 189
different-size data items, 525
discriminant, 178
division operator (/), 35, 38
DOS operating system, 552, 553
do statement, 235
double, 45
double precision numbers, 31, 112, 149
double quotes, to enclose strings, 18
driver function, 13
dynamic storage allocation, 505–11

E

EBCDIC (Extended Binary Coded Decimal Interchange Code), 33
echo printing, 193
editor program, 554–56
element of array, 242
end of file (EOF) marker, 210,332–33
end of string marker, 300, 303, 327, 427
ENTER key, 119, 454, 456, 463, 479
enum, 535
enumerated data types, 535–38
equality operator (==), 145, 149
equivalence statement, 123
errors
complie-time, 189
logic, 190
run-time, 189
syntax, 189
ESCAPE key, 121
escape sequences, 34
exchange sort, 271–74
exclusive OR operator (^), 522–24
exponential notation, 32
expression
conditional, 533–34
definition of, 35, 37, 156, 531
floating point, 37
integer, 37
mixed-mode, 37
relational, 144–46
simple, 144
value of, 145, 531
Extended Binary Coded Decimal Interchange Code (EBDIC), 33
extern, 310
external sorts, 269
external variable, 304, 314
extern storage classes, 314–16

F

fclose() function, 328, 354
fgets() function, 329, 334, 354
field width specifiers, 67, 475
file

access, 337
binary, 348, 350
master, 343
names, passing & returning, 446
organization, 337
text, 348
transaction, 343
update, 343
first-level structure diagram, 80
fixed increment iterations, 370–77
float, 45
floating point numbers, 31, 149
formatted, 68
flowchart, 6–7
flow of control, 144
fopen() function, 325
for loop
break statement in, 212,220
continue statement in, 212, 220
scanf() function within, 225
in single-dimensional array, 245
formal arguments, 278
format strings, 477
format modifies, 70
formatted output, 67
formatting strings, 475
for loop techniques
evaluating functions, 226
interactive input, 225
interactive loop control, 229
selection within, 226
for statement, 216–17
and input and output of arry values, 245
nested loops, 231–33
scanf() within a for loop, 225
FORTRAN (FORmula TRANslation), 3
fprintf() function, 327, 333, 354
fputs() function, 327,354
free() function, 506, 511
fscanf() function, 329, 333, 354
fseek() funciton, 337–39, 354
ftell() function, 337–39, 354
functions(s), 10–12, 24
argument declarations, 278
arguments of, 16, 36, 278, 281
called, 276
calling, 276
declarator, 278
fclose(), 328, 354
fopen(), 325
free(), 506, 511
getc(), 329, 354
getchar(), 286, 329, 333, 354, 454
gets(),
header line, 278
main(), 13–14, 20,24
malloc(), 506–11
printf(), 16–19, 24, 477
prototype, 277, 280
putchar(), 286, 333, 354, 454, 459
puts(), 333, 454, 455
scanf(), 107–12, 456
sqrt(), 98–99
standard library, 285–288
function body, 14, 278
function declaration, 280–85
function declarator, 278
function header, 277
function header line, 277
function names, 11, 12
parentheses in, 16, 24
function prototype, 280

G

getc() function, 329, 354
getchar() function, 286, 329, 333, 354, 454
gets() function, 333, 354, 454–55
global arrays, 250, 254
global scope, 304
global variable, 304
misuse of, 307
global variable storage classes, 313–16
goto statement, 548

H

header files, 17
header line, 278
hexadecimal numbers, 71–72, 355, 514

I

if-else chain, 162–67
if-else statement, 150–52
one-way selection, 154–55
replacement of, with conditional expressions, 533–34
#include statement, 17, 122
inclusive OR operator, 520–22
increment operator (++), 60–61, 137, 466
indexed variable, 243–47
index value, 243–47
indirect addressing, 96, 415
indirection operator (*), 94, 97, 414, 417, 429, 501
infinite loop, 198

initialization
 of arrays, 250–52, 472
 of pointer, 438
 run-time, 311
 of static variables, 312
 of structures, 484, 489
 of variables, 48
in-memory string conversions, 476–77
inner loop, 231
input/output (I/O), 75
 of array values, 245–48
input/output library functions, 286
input/output specifications, 75
int, 45
integer division, 38
integer values, 31
internal sorts, 269
interpreted language, 8
interpreter, 8
isalpha() function, 467
isdigit() function, 467
islower() function, 467
isupper() function, 467

K

Keyword, 11, 44

L

label, 548
larger-dimensional arrays, 257–58
left justification, 70
left shift operator (<<), 526
library functions, 2, 97, 285
linked lists, 498–504
linker, 556
lists, linked, 498–505
literal data, 122
local scope, 304
local variables, 281, 304
local variable storage classes, 310–11
logical file pointer, 332
logical operators, 146–48
 AND (&&), 146
 NOT (!), 147
 OR (||), 146
logical shift, 528
logic errors, 190
login procedures, 552
loops
 infinite, 198
 inner, 231
 nested, 231–34
 outer, 231
lvalue, 92, 411

M

machine language, 7
macro expansion, 540
macros, defining, 539–42
macro substitution, 540
magic numbers, 122
main() function, 13–14, 20, 24
 declaration statement within, 45
 as driver, 13
 structure of, 14, 22, 278
malloc() function, 506–11
mask, 519
master file, 343
master file update, 343–45
mathematical library functions, 97–103
midpoint method, 392
memory address, 28, 43
member operator (.), 482, 503
members of the structure, 482
memory, dynamic allocation of, 505–11
message, 36
Microsoft's Quick C compiler, 563–66
mnemonics, 12, 44
modularity, 10–14
 top-down design in, 77–80
modular programs, 4, 10
modules, 10
modulus operator (%), 38
multiplication operator(*), 35, 39

N

named constants, 122–25
negation, 38
negation operator (−), 38
nested for statements, in
 multidimensional arrays, 256
nested if statements, 161–71
 errors with, 186–87
 if-else chain, 162–63
nested loops, 231–33
newline escape sequence (\n), 19, 24, 34, 256, 302, 329–30, 334
nonprintable characters, 355
NOT operator (!), 147
null character, 252, 332, 454, 544
NULL pointer, 500, 504
null statement, 213
number bases, 71–73

number codes, 27
number sign (#), 17, 122
numerical accuracy, 149
numerical integration, 388–402
numerical results, displaying, 36–37, 67–70

O

object program, 8, 555
octal character codes, 71, 355
octal numbers, 71, 355, 517, 518
offsets, 429
one-dimensional array, 241–48
one-way if statement, 154–55
one-way selection, 154–55
opening a file, 324–25
operand, 35, 144
operating system, 553
operator
 addition, 35, 39
 address (&), 93, 109, 412, 428
 arithmetic, 35–39, 102–03
 assignment, 54–57
 binary, 517
 bit, 517–28, 533
 conditional (?:), 533–34
 decrement(−−), 61
 division, 35, 39
 equality (==), 145, 149
 increment (++) 60, 61, 137
 indirection (*), 94, 97,414, 417, 429, 501
 left shift, 526
 logical, 146–48
 modulus, 38
 multiplication, 35, 39
 negation, 38, 39
 postfix decrement (−−), 61
 postfix increment (++), 61
 precedence of, 38–39, 148, 532, 574
 prefix increment, 61
 prefix decrement, 61
 relational, 144–46
 remainder, 39
 right shift, 526
 shift, 526–28
 sizeof (), 31, 95
 subtraction, 35, 39
 ternary, 534
 urnary, 38
OR operator (||), 146
outer loop, 231
output, formatted, 67

P

parameter, 278
parentheses
 in arithmetic expressions, 38
 in for statement, 216
 in function, 16, 24
 in function names, 11, 16
 in macro arguments, 542
 in main() function, 20–21
Pascal, 4
Pascal, Blaise, 4
passing addresses, 421–26
passing arrays, 291–97
passing file names, 446–49
passing structures, 491–95
pendulum clocks, 83–85
percent sign (%), 36–37
physical file pointer, 332
pointer(s), 94, 414
 advanced notation, 442–45
 array names as, 428–34
 declaring, 96, 415
 decrementing, 435–36
 incrementing, 435–36
 and library functions, 461–67
pointer arithmetic, 436–38
pointer arrays, 471–74
 string definitions, 468–74
pointer declarations, 96, 415
pointer initialization, 438
pointer variables, 94, 414
pollen counts, 340–43
populating the structure, 482
postfix decrement (−−) operator, 61
postfix increment (++) operator, 61, 495
pound sign (#), 17, 122
precedence, 38–39, 148, 532, 574
prefix decrement operator, 61
prefix increment operator, 61
PRIME operating system, 552, 553, 554, 556, 557, 558
printable characters, 354
printf() function, 16–19, 24, 477
 conversion control sequences in, 35, 47, 93–94, 412–13
 and conversion characters, 36
 in debugging, 193
 program costs, 319–22
 display of characters, 71–73
 display of floating point variables, 36
 display of integer values, 37, 72
 display of unsigned integers, 94fn
 in evaluating arithmetic expressions, 36–37
 full-width specifiers in, 67–71, 475–76

and input and output of array values, 246
and newline escape sequences in, 19
program costs, 319–22
program development, 74–81, 138–42
analysis phase, 74–75, 138–39
coding phase, 76–77, 138–39
design phase, 75–76, 138–39
testing phase, 77, 138–39, 191–93
program loop, 195–96
programming, 2, 138
programming errors, common, 23–24, 90–91, 136–37, 185–87, 238–39, 268, 317–18, 352–53, 407–08, 449–51, 478–79, 514–15
programming language, 2
programming style, 20–22
program tracing, 193
program translation, 6
program verification and testing, 189
prompt 109
pseudocode, 6
putchar() function, 286, 333, 354, 454, 459
puts() function, 300, 301, 380, 391

Q

quadratic equations, 177–80

R

random file access, 337–46
reading files, 327–33
record, 340
rectangular approximations, 388–95
register storage class, 313
registers, 313
relational expressions, 144–46
logical operators, 146–48
simple, 144
value of, 145
relational operators, 144–46, 434, 533
remainder operator (%), 39
return, 279
returning file names, 446–49
returning structures, 495–97
returning values, 277–78
rewind() function, 337
right shift operator (>>), 526
root finding, 369–87
run-time errors, 189
run-time initialization, 312
rvalue, 92, 411

S

scalar variables, 241
scanf() function, 107–12, 456
with buffered input, 119–21
and input and output of array values, 245
passing addresses to, 111–12
use of, to input string data, 454–56
within a for loop, 225–26
within a while loop, 203–08
break statement, 212
continue statement, 212–213
null statement, 213
scope, 304
global, 304
local, 304
secant method, 381
selection sort, 269–71
selection statements, 144
semicolon
as null statement, 381
in assignment statement, 53–55
in for statement, 216
lack of, in # define statement, 124
in main() function, 14, 17
to terminate statement, 14, 17
sentinels, 208–11
sequential control, 144
sequential organization, 337
shift operators, 526–28
signed numbers, 27–28, 526
simple arithmetic expression, 35
Simpson's method, 398–402
simultaneous linear equations, 358–66
sine approximation, 103–34
single-dimensional arrays, 241–48
single precision numbers, 31
single quotes, 33
single structures, 481–86
sizeof() operator, 31, 95
sort(s)
bubble, 269, 272
exchange, 271–74
internal, 269
selection, 269–71
source program, 7, 554, 556
SPACE key, 121
sprintf() function, 475–76
sqrt() function, 98–99
sscanf() function, 475–77
standard device files, 332–34
standard input file, 332–33

standard library functions, 285–88
See also Library functions
input/output library functions, 286
mathematical library functions, 97–103
miscellaneous routines, 287–88
string library functions, 286
standard output file, 333
statement(s)
assignment, 44, 53–55
break, 212
compound, 152–53
continue, 212
counting, 60
declaration, 45–48
#define, 122–24, 210
do, 235
equivalence, 123
for, 216–17
goto, 548
if-else, 150–52
#include, 17, 122
null, 213
selection, 144
switch, 172–75
typedef, 538–39
while, 195–201
static array, 251
static storage class, 310
stdio.h, 17
storage allocation, dynamic, 505–11
storage size, determining, 95
strcat() function, 467
strchr() function, 467
strcmp() function, 467
strcopy() function, 457, 461
while statement in, 462
strcpy() function, 467
string constant, 454
string, 18, 454
definition of, 18, 454, 468, 479
format, 477
formatting, 475
in-memory conversions, 476–77
input and output, 454–56
character-by-character, 458–61
library functions, 286
and pointer arrays, 468–74
processing, 456–58
strlen() function, 467
structure(s), 481–87
arrays of, 487–89
content of, 481
data types in, 483–84, 488
declaration of, 481–82
external, 484
initialization of, 448
members of, 482
passing, 491–95
populating, 482
returning, 495–97
single, 481–86
static, 484
tag name for, 483, 485
template for, 483
structure pointer (–>) operator, 503
structured programming, 4
structure type, 483
subscript, 243–44
value of, 243
subscripted variable, 243–44
subtraction, 35, 39
subtraction operator (–), 35, 39
switch statement, 172–75
symbolic constants, 123
symbolic name, 122
syntax error, 189–90

T

tag name, 49, 483, 485, 513
telephone networks, 85–87
template, 483
ternary operator, 534
text mode, 556
tolower() function, 467
top-down program development, 74–81, 139
modularity, 77–80
toupper() function, 467
transaction file 343
trupazoidal approximations, 395
truncated, 38
two-dimensional arrays, 253–56, 442–43
two's complement numbers, 27–28
type conversions 102–03
typedef statement, 538–39

U

unary operators, 38
union, 512–13
UNIX operating system, 210, 543, 552, 553, 554, 555, 557, 558
user-specified data types, 535–39

V

validity checks, 236
value box, 27–28
value box conversion method, 27–28
variable(s), 44–45, 245
  address of, 43–44, 92, 411, 429
  external, 304, 314
  global, 304
  indexed, 243–44
  local, 281, 304
  scalar, 241
  subscripted 243–44
variable name, 44
variable scope, 304–08
variable storage class, 310–16
VAX-VMS operating system, 552, 553, 554, 556, 557, 558
void, 279
void data type, 279

W

whole loop, 196–201
  scanf() function in, 203–07
while statement, 195–201
  in strcopy(), 462
white space, 21, 250
words, 28
writing files, 327–33